Zoltan: Data-Management Services for Parallel Applications

User's Guide

The Zoltan Team
Sandia National Laboratories:
Erik Boman
Karen Devine
Lee Ann Fisk
Robert Heaphy
Bruce Hendrickson
Courtenay Vaughan

Ohio State University
Umit Catalyurek
Doruk Bozdag

National Institute of Standards and Technology
William F. Mitchell

Zoltan User's Guide, Version 2.01

[Zoltan Home Page | Next: Introduction]

Introduction

Project Motivation

Over the past decade, parallel computers have been used with great success in many scientific simulations. While differing in their numerical methods and details of implementation, most applications successfully parallelized to date are "static" applications. Their data structures and memory usage do not change during the course of the computation. Their inter-processor communication patterns are predictable and non-varying. And their processor workloads are predictable and roughly constant throughout the simulation. Traditional finite difference and finite element methods are examples of widely used static applications.

However, increasing use of "dynamic" simulation techniques is creating new challenges for developers of parallel software. For example, adaptive finite element methods refine localized regions the mesh and/or adjust the order of the approximation on individual elements to obtain a desired accuracy in the numerical solution. As a result, memory must be allocated dynamically to allow creation of new elements or degrees of freedom. Communication patterns can vary as refinement creates new element neighbors. And localized refinement can cause severe processor load imbalance as elemental and processor work loads change throughout a simulation.

Particle simulations and crash simulations are other examples of dynamic applications. In particle simulations, scalable parallel performance depends upon a good assignment of particles to processors; grouping physically close particles within a single processor reduces inter-processor communication. Similarly, in crash simulations, assignment of physically close surfaces to a single processor enables efficient parallel contact search. In both cases, data structures and communication patterns change as particles and surfaces move. Re-partitioning of the particles or surfaces is needed to maintain geometric locality of objects within processors.

We developed the Zoltan library to simplilfy many of the difficulties arising in dynamic applications. Zoltan is a collection of data management services for unstructured, adaptive and dynamic applications. It includes a suite of parallel partitioning algorithms, data migration tools, distributed data directories, unstructured communication services, and dynamic memory management tools. Zoltan's data-structure neutral design allows it to be used by a variety of applications without imposing restrictions on application data structures. Its object-based interface provides a simple and inexpensive way for application developers to use the library and researchers to make new capabilities available under a common interface.

The Zoltan Toolkit

The Zoltan Library contains a number of tools that simplify the development and improve the performance of parallel, unstructured and adaptive applications. The library is organized as a toolkit, so that application developers can use as little or as much of the library as desired. The major packages in Zoltan are listed below.

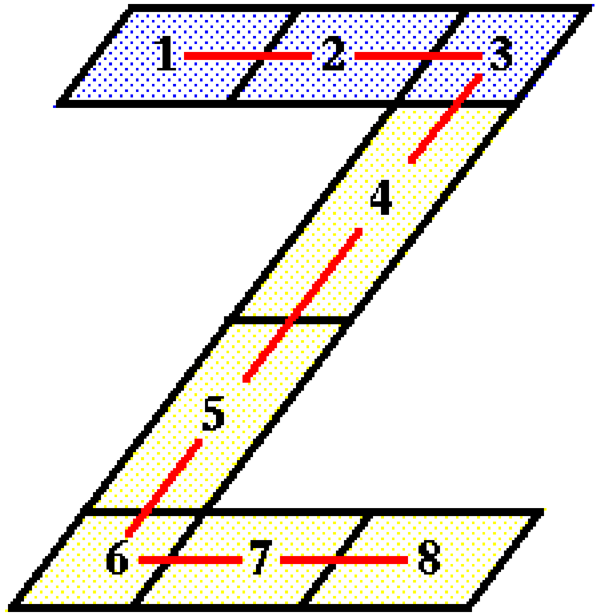
- A suite of dynamic load balancing and parallel repartitioning algorithms; switching between algorithms is easy, allowing straightforward comparisons of algorithms in applications.
- Data migration tools for moving data from old partitions to new one.
- Distributed data directories: scalable (in memory and computation) algorithms for locating needed off-processor data.
- An unstructured communication package that insulates users from the details of message sends and receives.
- Dynamic memory management tools that greatly simplify dynamic memory debugging on state-of-the-art parallel computers.
- A sample application *zdrive*. It allows algorithm developers to test changes to Zoltan without having to run Zoltan in a large application code. Application developers can use the *zdrive* code to see examples of function calls to Zoltan and the implementation of query functions.

Terminology

Our design of Zoltan does not restrict it to any particular type of application. Rather, Zoltan operates on uniquely identifiable data items that we call *objects*. For example, in finite element applications, objects might be elements or nodes of the mesh. In particle applications, objects might be particles. In linear solvers, objects might be matrix rows.

Each object must have a unique *global identifier (ID)* represented as an array of unsigned integers. Common choices include global numbers of elements (nodes, particles, rows, and so on) that already exist in many applications, or a structure consisting of an owning processor number and the object's local-memory index. Objects might also have local (to a processor) IDs that do not have to be unique globally. Local IDs such as addresses or local-array indices of objects can improve the performance (and convenience) of Zoltan's interface to applications.

We use a simple example to illustrate the above terminology. In the figure below, a simple finite element mesh is presented.



The blue and yellow shading indicates the mesh is partitioned for two processors. An application must provide information about the current mesh and partition to Zoltan. If, for example, the application wants Zoltan to perform operations on the elements of the mesh, it must provide information about the elements when Zoltan asks for object information.

In this example, the elements have unique numbers assigned to them, as shown by the numbers in the elements. These unique numbers can be used as global IDs in Zoltan. In addition, on each processor, local numbering information may be available. For instance, the elements owned by a processor may be stored in arrays in the processor's memory. An element's local array index may be provided to Zoltan as a local ID.

For geometric algorithms, the application must provide coordinate information to Zoltan. In this example, the coordinates of the mid-point of an element are used.

For graph-based algorithms, information about the connectivity of the objects must be provided to Zoltan. In this example, the application may consider elements connected if they share a face. The connections between elements, or *edges* of the connectivity graph, are shown in red. Connectivity information is passed to Zoltan by specifying a neighbor list for an object. The neighbor list consists of the global IDs of neighboring objects and the processor(s) currently owning those objects.

The table below summarizes the information provided to Zoltan by an application for this finite element mesh. Information about the objects includes their global and local IDs, geometry data, and graph data.

	Object IDs		Geometry Data	Graph Data	
Processor	Global	Local	(coordinates)	Neighbor Global ID List	Neighbor Processor List
Blue	1	0	(0.8,2.9)	2	Blue
	2	1	(1.7,2.9)	1,3	Blue,Blue
	3	2	(2.5,2.9)	2,4	Blue,Yellow
Yellow	4	0	(2.0,2.1)	3,5	Blue,Yellow
	5	1	(1.1,1.0)	4,6	Yellow,Yellow
	6	2	(0.5,0.2)	5,7	Yellow,Yellow

	7	3	(1.3,0.2)	6,8	Yellow,Yellow
	8	4	(2.1,0.2)	7	Yellow

Zoltan Design

To make Zoltan easy to use, we do not impose any particular data structure on an application, nor do we require an application to build a particular data structure for Zoltan. Instead, Zoltan uses a callback function interface, in which Zoltan queries the application for needed data. The application must provide simple functions that answer these queries.

To keep the application interface simple, we use a small set of callback functions and make them easy to write by requesting only information that is easily accessible to applications. For example, the most basic partitioning algorithms require only four callback functions. These functions return the number of objects owned by a processor, a list of weights and IDs for owned objects, the problem's dimensionality, and a given object's coordinates. More sophisticated graph-based partitioning algorithms require only two additional callback functions, which return the number of edges per object and edge lists for objects.

[Table of Contents | Next: Zoltan Release Notes | Previous: Table of Contents]

Release Notes

Release notes are available for the following releases of Zoltan:

Zoltan Release Notes v2.01

Zoltan v2.01 includes enhancements to version 2.0.

- F90 interface fixes to comply with standard F90 (e.g., shortened variable names and continuation lines). The hypergraph callback function names have changed, but C and C++ compatibility with v2.0 is maintained.
- Performance improvement to initial building of hypergraphs from application data.
- Major bug fix for dense-edge removal in parallel hypergraph method; partitioning of hypergraphs with edges containing more than 25% of the vertices was affected by this bug.
- Minor fixes to parallel hypergraph code.

Zoltan Release Notes v2.0

Zoltan v2.0 includes several major additions:

- Parallel hypergraph partitioning.
- Parallel graph coloring, both distance-1 and distance-2.
- Multicriteria geometric partitioning (RCB).
- C++ interface.

Zoltan Release Notes v1.54

Some versions of MPICH have a bug in MPI_Reduce_scatter; they can report errors with MPI_TYPE_INDEXED. In Zoltan v1.54's unstructured communication package, calls to MPI_Reduce_scatter have been replaced with separate calls to MPI_Reduce and MPI_Scatter.

Zoltan Release Notes v1.53

Zoltan v1.53 includes the following new capabilities:

- Portability to BSD Unix and Mac OS X was added.
- Averaging of RCB and RIB cuts was added; see Zoltan parameter AVERAGE_CUTS.
- A new function **Zoltan_RCB_Box** returns information about subdomain bounding boxes in RCB decompositions.
- F90 interface to **Zoltan_Order** was added.
- Warnings that load-imbalance tolerance was not met are no longer printed when DEBUG_LEVEL == 0.
- Minor bugs were addressed.

Zoltan Release Notes v1.52

Zoltan v1.52 includes the following new capabilities:

- List-based graph callback functions ZOLTAN_NUM_EDGES_MULTI_FN and ZOLTAN_EDGE_LIST_MULTI_FN were added to mirror support and performance given by the list-based geometric function ZOLTAN_GEOM_MULTI_FN.
- Support for ParMETIS v3.1 was added.
- Minor bugs were addressed.

Zoltan Release Notes v1.5

This section describes improvements to Zoltan in Version 1.5. Every attempt was made to keep Zoltan v1.3 backwardly compatible with previous versions. Users of previous versions of Zoltan should refer to the Backward Compatibility Notes.

Short descriptions of the following features are included below; follow the links for more details.

Partition Remapping

During partitioning, Zoltan v1.5 can renumber partitions so that the input and output partitions have greater overlap (and, thus, lower data-migration costs). This remapping is controlled by Zoltan parameter *REMAP*. Experiments have shown that using this parameter can greatly reduce data migration costs.

Unequal Numbers of Partitions and Processors

Zoltan v1.5 can be used to generate *k* partitions on *p* processors, where *k* is not equal to *p*. Function **Zoltan_LB_Partition** (replacing **Zoltan_LB_Balance**) can generate arbitrary numbers of partitions on the given processors. The number of desired partitions is set with parameters *NUM_GLOBAL_PARTITIONS* or *NUM_LOCAL_PARTITIONS*. Both partition and processor information are returned by **Zoltan_LB_Partition**, **Zoltan_LB_Box_PP_Assign**, and **Zoltan_LB_Point_PP_Assign**. New Zoltan query functions **ZOLTAN_PARTITION_FN** and **ZOLTAN_PARTITION_MULTI_FN** return objects' partition information to Zoltan. **Zoltan_LB_Balance** can still be used for *k* equal to *p*.

Non-Uniform Partition Sizes

Partition sizes for local and global partitions can be specified using **Zoltan_LB_Set_Part_Sizes**, allowing non-uniformly sized partitions to be generated by Zoltan's partitioning algorithms.

Zoltan Interface Updated

To support the concept of partitions separate from processors, many new interface functions were added to Zoltan v1.5 (e.g., **Zoltan_LB_Partition** and **Zoltan_Migrate**). These functions mimic previous Zoltan functions (e.g., **Zoltan_LB_Balance** and **Zoltan_Help_Migrate**, respectively), but include both partition and processor information. Both the new and old interface functions work in Zoltan v1.5. See the notes on Backward Compatibility.

Robust HSFC Box Assign

Function **Zoltan_LB_Box_PP_Assign** now works for the Hilbert Space-Filling Curve algorithm (HSFC), in addition to the RCB and RIB algorithms supported in previous versions of Zoltan. **Zoltan_LB_Point_PP_Assign** continues to work for HSFC, RCB and RIB.

Matrix Ordering

Zoltan v1.5 contains a matrix-ordering interface **Zoltan_Order** to ParMETIS' matrix-ordering functions. New graph-based matrix-ordering algorithms can be easily added behind this interface.

Performance Improvements

Many performance improvements were added to Zoltan v1.5.

- List-based callback functions have been added to Zoltan (**ZOLTAN_GEOM_MULTI_FN**, **ZOLTAN_PARTITION_MULTI_FN**, **ZOLTAN_OBJ_SIZE_MULTI_FN**, **ZOLTAN_PACK_OBJ_MULTI_FN**, and **ZOLTAN_UNPACK_OBJ_MULTI_FN**); these functions allow entire lists of data to be passed from the application to Zoltan, replacing per-object callbacks.
- **Zoltan_Migrate** now can accept either import lists, export lists, or both. It is no longer necessary to call **Zoltan_Invert_Lists** or **Zoltan_Compute_Destinations** to get appropriate input for **Zoltan_Migrate**.
- Zoltan v1.5 contains performance improvements within individual algorithms. We recommend users upgrade to the latest version.

Bug Fixes

Bug fixes were made to Zoltan's algorithms and interface. Users of previous versions of Zoltan are encouraged to upgrade.

Zoltan Release Notes v1.3

This section describes improvements to Zoltan in Version 1.3. Every attempt was made to keep Zoltan v1.3 backwardly compatible with previous versions. Users of previous versions of Zoltan should refer to the Backward Compatibility Notes.

Short descriptions of the following features are included below; follow the links for more details.

More Data Services

Zoltan's mission has been widened beyond its original focus on dynamic load-balancing algorithms. Now Zoltan also provides data management services to parallel, unstructured, and adaptive computations. Several packages of parallel data services have been added and made available to application developers.

These services include the following:

- An unstructured communication package that simplifies complicated communication by insulating applications from the details of message sends and receives.
- A distributed data directory that allows applications to efficiently (in memory and time) locate off-processor data.
- A dynamic memory management package that simplifies debugging of memory allocation problems on state-of-the-art parallel computers.

New Hilbert Space-Filling Curve Partitioning

Zoltan now includes a fast, efficient implementation of Hilbert Space-Filling Curve (HSFC) partitioning. This geometric method also includes support for Zoltan_LB_Box_Assign and Zoltan_LB_Point_Assign functions.

Support for Structured-Grid Partitioning

Zoltan's Recursive Coordinate Bisection (RCB) partitioning algorithm has been enhanced to allow generation of strictly rectilinear subdomains. This capability can be used for partitioning of grids for structured-grid applications. See parameter *RCB_RECTILINEAR_BLOCKS*.

Support for ParMETIS v3.0

In addition to providing interfaces to ParMETIS v2.0 and PJostle, Zoltan now provides an interfaces ParMETIS v3.0. Full support of ParMETIS v3.0's multiconstraint and multiobjective partitioning is included.

Performance Improvements

Performance of Zoltan's partitioning algorithms has been improved through a number of code optimizations and new features. In addition, user parameter *RETURN_LISTS* can be used to specify which returned arguments are computed by **Zoltan_LB_Balance**, allowing reduced work in partitioning. In the Recursive Coordinate Bisection (RCB) partitioning algorithm, user parameters allow cut directions to be locked in an attempt to minimize data movement; see parameters *RCB_LOCK_DIRECTIONS* and *RCB_SET_DIRECTIONS*.

Zoltan Interface Updated

Zoltan has adopted a more modular design, making it easier to use by applications and easier to modify by algorithm developers. Names in the Zoltan interface and code are tied more closely to their functionality. Full backward compatibility is supported for users of previous versions of Zoltan.

Improved Test Suite

The Zoltan test suite has been improved, with more tests providing greater code coverage and platform-specific answer files accounting for differences due to computer architectures.

Bug Fixes

Some bug fixes were made to Zoltan's algorithms and interface. Users of previous versions of Zoltan are encouraged to upgrade.

[Table of Contents | Next: Using the Zoltan Library | Previous: Introduction]

Using the Zoltan library

This section contains information needed to use the Zoltan library with applications:

System requirements.
Data types for global and local IDs.
Instructions for building the Zoltan library.
Instructions for building applications that use Zoltan.

System Requirements

Zoltan was designed to run on parallel computers and clusters of workstations. In order to build and use Zoltan, you will need:

- ANSI C compiler.
- MPI library for message passing (version 1.1 or higher), such as MPICH or LAM.
- A Unix-like operating system (e.g., Linux or Solaris) and *gmake* (GNU Make) are recommended to build the library.
- A Fortran90 compatible compiler is required if you wish to use Zoltan with Fortran applications.

Zoltan has been tested on a variety of platforms, including Linux, Solaris, Irix, and the ASCI Red Teraflop machine. If you wish to use Zoltan on a non-Unix operating system, for example Windows NT or 2000, you will have to port Zoltan yourself.

Data Types for Object IDs

Application query functions and application callable library functions use global and local identifiers (IDs) for objects. *All objects to be used in load balancing must have unique global IDs.* Zoltan stores an ID as an array of unsigned integers. The number of entries in these arrays can be set using the NUM_GID_ENTRIES and NUM_LID_ENTRIES parameters; by default, one unsigned integer represents an ID. Applications may use whatever format is most convenient to store their IDs; the IDs can then be converted to and from Zoltan's ID format in the application-registered query functions.

The following type definitions are defined in *include/zoltan_types.h*; they can be used by an application for memory allocation, MPI communication, and as arguments to load-balancing interface functions and application-registered query functions.

typedef unsigned int **ZOLTAN_ID_TYPE**;
typedef **ZOLTAN_ID_TYPE** ***ZOLTAN_ID_PTR**;
#define **ZOLTAN_ID_MPI_TYPE** MPI_UNSIGNED

In the Fortran interface, IDs are passed as arrays of integers since unsigned integers are not supported in Fortran. See the description of the Fortran interface for more details.

The local IDs passed to Zoltan are not used by the library; they are provided for the convenience of the application and can contain any information desired by the application. For instance, local array indices for objects may be passed as local IDs, enabling direct access to object data in the query function routines. See the application-registered query functions for more details. The source code distribution contains an example application *zdrive* in which global IDs are integers and local IDs are local array indices. One may choose not to use local ids at all, in which case NUM_LID_ENTRIES may be set to zero.

Some Zoltan routines (e.g., **Zoltan_LB_Partition** and **Zoltan_Invert_Lists**) allocate arrays of type **ZOLTAN_ID_PTR** and return them to the application. Others (e.g., **Zoltan_Order** and **Zoltan_DD_Find**) require the application to allocate memory for IDs. Memory for IDs can be allocated as follows:

```
ZOLTAN_ID_PTR gids;
int num_gids, int num_gid_entries;
gids = (ZOLTAN_ID_PTR) ZOLTAN_MALLOC(num_gids *
num_gid_entries * sizeof(ZOLTAN_ID_TYPE);
```

The system call *malloc* may be used instead of **ZOLTAN_MALLOC**.

Building the Zoltan Library

The Zoltan library is implemented in ANSI C and can be compiled with any ANSI C compiler. Makefiles are included with the source code; these makefiles require the GNU Make (gmake) utility. The top-level Makefile defines targets for the Zoltan library, test driver programs in C, C++ and Fortran90, and two graphical utilities useful for visualization of geometric partitions. (The test drivers and utilities are primarily intended for use by developers.) This Makefile need not be edited to build Zoltan. Instead, environment-specific definitions are specified in the configuration file, *Utilities/Config/Config.<platform>*, where *<platform>* specifies the particular platform for which Zoltan is being built. Paths to compilers, include files, and libraries are defined in this file and are then read by the top-level Makefile. Examples of configuration files for Solaris, Sandia's ASCI Red (tflop) computer, SGI workstations, and PCs running Linux are included in the *Utilities/Config* subdirectory. A well-commented version of the configuration file, *Utilities/Config/Config.generic*, is also included; this file can be used as a template for new environment-specific files. The variables in these files should be edited to reflect the new system's environment.

The command for building Zoltan is shown below:

gmake [options] zoltan

where the options that may be specified are listed below.

Options to gmake:

ZOLTAN_ARCH=<platform>	Specify the target architecture for the Zoltan library. A corresponding file, *Utilities/Config/Config.<platform>*, containing environment definitions for *<platform>*, must be created in the *Utilities/Config* directory.

YES_FORTRAN=1 — Include Fortran support in the Zoltan library. By default, the Zoltan library is built without the interface that allows use from Fortran applications. If this option is specified, the Fortran interface is compiled and included in the library. Use of this option requires that a Fortran 90 (or 95, or later) compiler is available.

As an alternative to typing the options on the gmake command line, they may be set as environment variables; e.g., if you are using a C-shell (csh or tcsh), type

setenv ZOLTAN_ARCH <*platform*>

or if you are using a Bourne-type shell (e.g., sh or bash), type

ZOLTAN_ARCH = <*platform*>*;* export ZOLTAN_ARCH

The resulting library *libzoltan.a*, object files, and dependency files are stored in the directory Obj_<*platform*>.

Testing the Zoltan Library

The *examples* directory contains simple C and C++ examples which use the Zoltan library. The Makefile in this directory has three targets:

gmake ZOLTAN_ARCH=<platform> C_Examples

This builds simple C language examples that use the Zoltan library to perform load balancing.

gmake ZOLTAN_ARCH=<platform> CPP_Examples

This builds simple C++ language examples that use the Zoltan library to perform load balancing. To build C++ applications, define CPPC to point to your C++ compiler in the Config.<platform> file.

gmake ZOLTAN_ARCH=<platform> all

Build both C and C++ examples. Don't forget to define CPPC in your Config.<platform> file.

Some of these examples make use of a small library of support routines found in the *examples/lib* directory. These routines create simple test meshes of varying sizes, perform error checking across the parallel application, and define Zoltan call backs.

The "right" answer for these tests depends on the number of processes with which you run the tests. In general, if they compile successfully, run quickly (in seconds), and produce reasonable looking output, then you have been successful in building Zoltan.

Building Applications that use Zoltan

The C library interface is described in the include file *include/zoltan.h*; this file should be included in all C application source files that call Zoltan library routines.

The C++ interface to Zoltan is implemented in header files which define classes that wrap the Zoltan C library. The file *include/zoltan_cpp.h* defines the **Zoltan** class which encapsulates a load balancing data structure and the Zoltan load balancing functions which operate upon it. Include this header file instead in your C++ application. Note that C++ applications should call the C function **Zoltan_Initialize** before creating a **Zoltan** object.

Fortran applications must USE module zoltan and specify *Zoltan/Obj_<platform>* as a directory to be searched for module information files.

The C, C++ or Fortran application should then be linked with the Zoltan library (built with Fortran support in the Fortran case) and its utility libraries by including

-lzoltan

in the linking command for the application. Communication within Zoltan is performed through MPI, so appropriate MPI libraries must be linked with the application. Third-party libraries, such as ParMETIS and Jostle, must be also be linked with the application if they were included in compilation of the Zoltan library. (A courtesy copy of ParMETIS is included with the Zoltan distribution; Jostle must be obtained directly from http://www.gre.ac.uk/~jjg01/.)

For applications that used versions of Zoltan before Zoltan v.1.3, only minor updates to the application build process are needed; see the section on backward compatibility of Zoltan.

[Table of Contents | Next: Zoltan Interface Functions | Previous: Zoltan Release Notes]

Zoltan Interface Functions

An application calls a series of dynamic load-balancing library functions to initialize the load balancer, perform load balancing and migrate data. This section describes the syntax of each type of interface function:

Examples of the calling sequences for initialization, load-balancing, and data migration are included in the Initialization, Load-Balancing, and Migration sections, respectively, of the Examples of Library Usage.

Error Codes

All interface functions, with the exception of **Zoltan_Create**, return an error code to the application. The possible return codes are defined in *include/zoltan_types.h* and Fortran module zoltan, and are listed in the table below.

Note: Robust error handling in parallel has not yet been achieved in Zoltan. When a processor returns from Zoltan due to an error condition, other processors do not necessarily return the same condition. In fact, other processors may not know that the original processor has returned from Zoltan, and may wait indefinitely in a communication routine (e.g., waiting for a message from the original processor that is not sent due to the error condition). The parallel error-handling capabilities of Zoltan will be improved in future releases.

ZOLTAN_OK	Function returned without warnings or errors.
ZOLTAN_WARN	Function returned with warnings. The application will probably be able to continue to run.
ZOLTAN_FATAL	A fatal error occured within the Zoltan library.
ZOLTAN_MEMERR	An error occurred while allocating memory. When this error occurs, the library frees any allocated memory and returns control to the application. If the application then wants to try to use another, less memory-intensive algorithm, it can do so.

Return codes defined in include/zoltan_types.h.

Naming conventions

The C, Fortran and C++ interfaces follow consistent naming conventions, as illustrated in the following table.

	C and Fortran	**C++**
Partitioning and migration functions example: perform partitioning example: assign a point to a partition	**Zoltan_***function*() **Zoltan_LB_Partition()** **Zoltan_LB_Point_Assign()**	**Zoltan::***function()* **Zoltan::LB_Partition()** **Zoltan::LB_Point_Assign()**
Unstructured communication example: perform communication	**Zoltan_Comm_***function* **Zoltan_Comm_Do()**	**Zoltan_Comm::***function* **Zoltan_Comm::Do()**
Distributed data example: find objects in a remote process	**Zoltan_DD_***function* **Zoltan_DD_Find()**	**Zoltan_DD::***function* **Zoltan_DD::Find()**
Timers example: print timing results	**Zoltan_Timer_***function* **Zoltan_Timer_Print()**	**Zoltan_Timer::***function* **Zoltan_Timer::Print()**

In particular, the C++ **Zoltan** class represents a load balancing instance and the methods that operate on it. The method name is identical to the part of the C and Fortran function name that indicates the function performed. A C++ **Zoltan_Comm** object represents an instance of unstructured communication, a C++ **Zoltan_DD** object represents a distributed directory, and a C++ **Zoltan_Timer** object is a timer. Their method names are derived similarly.

[Table of Contents | Next: Initialization Functions | Previous: Using the Library]

General Interface Functions

Functions used to initialize and manipulate Zoltan's data structures are described below:

- **Zoltan_Initialize**
- **Zoltan_Create**
- **Zoltan_Copy**
- **Zoltan_Copy_To**
- **Zoltan_Set_Param**
- **Zoltan_Set_Param_Vec**
- **Zoltan_Set_Fn**
- **Zoltan_Set_<*zoltan_fn_type*>_Fn**
- **Zoltan_Destroy**

C and C++:	int **Zoltan_Initialize** (int *argc*, char ***argv*, float **ver*);
FORTRAN:	FUNCTION **Zoltan_Initialize**(*argc*, *argv*, *ver*) INTEGER(Zoltan_INT) :: Zoltan_Initialize INTEGER(Zoltan_INT), INTENT(IN), OPTIONAL :: argc CHARACTER(LEN=*), DIMENSION(*), INTENT(IN), OPTIONAL :: argv REAL(Zoltan_FLOAT), INTENT(OUT) :: ver

---

The **Zoltan_Initialize** function initializes MPI for Zoltan. If the application uses MPI, this function should be called after calling **MPI_Init**. If the application does not use MPI, this function calls **MPI_Init** for use by Zoltan. This function is called with the *argc* and *argv* command-line arguments from the main program, which are used if **Zoltan_Initialize** calls **MPI_Init**. From C, if **MPI_Init** has already been called, the *argc* and *argv* arguments may have any value because their values will be ignored. From Fortran, if one of *argc* or *argv* is omitted, they must both be omitted. If they are omitted, *ver* does NOT have to be passed as a keyword argument.

Zoltan_Initialize returns the Zoltan version number so that users can verify which version of the library their application is linked to.

C++ applications should call the C **Zoltan_Initialize** function before using the C++ interface to the Zoltan library.

Arguments:

argc	The number of command-line arguments to the application.
argv	An array of strings containing the command-line arguments to the application.
ver	Upon return, the version number of the library.

Returned Value:

int	Error code.

C:	struct **Zoltan_Struct** ***Zoltan_Create** (MPI_Comm *communicator*);
FORTRAN:	FUNCTION **Zoltan_Create**(*communicator*) TYPE(Zoltan_Struct), pointer :: Zoltan_Create INTEGER, INTENT(IN) :: communicator
C++:	**Zoltan** (const MPI_Comm &*communicator* = MPI_COMM_WORLD);

The **Zoltan_Create** function allocates memory for storage of information to be used by Zoltan and sets the default values for the information. The pointer returned by this function is passed to many subsequent functions. An application may allocate more than one **Zoltan_Struct** data structure; for example, an application may use several **Zoltan_Struct** structures if, say, it uses different decompositions with different load-balancing techniques.

In the C++ interface to Zoltan, the **Zoltan** class represents a Zoltan load balancing data structure and the functions that operate on it. It is the constructor which allocates an instance of a **Zoltan** object. It has no return value.

Arguments:

communicator	The MPI communicator to be used for this Zoltan structure. Only those processors included in the communicator participate in Zoltan functions. If all processors are to participate, *communicator* should be **MPI_COMM_WORLD** .

Returned Value:

struct **Zoltan_Struct** *	Pointer to memory for storage of Zoltan information. If an error occurs, NULL will be returned in C, or the result will be a nullified pointer in Fortran. Any error that occurs in this function is assumed to be fatal.

C:	struct **Zoltan_Struct** ***Zoltan_Copy** (**Zoltan_Struct** **from*);
FORTRAN:	FUNCTION **Zoltan_Copy**(*from*) TYPE(Zoltan_Struct), pointer :: Zoltan_Copy TYPE(Zoltan_Struct), INTENT(IN) :: from
C++:	**Zoltan** (const Zoltan &zz);

The **Zoltan_Copy** function creates a new **Zoltan_Struct** and copies the state of the existing **Zoltan_Struct**, which it has been passed, to the new structure. It returns the new **Zoltan_Struct**.

There is no direct interface to **Zoltan_Copy** from C++. Rather, the **Zoltan** copy constructor invokes the C library **Zoltan_Copy** program.

Arguments:	
from	A pointer to the **Zoltan_Struct** that is to be copied.
Returned Value:	
struct **Zoltan_Struct** *	Pointer to a new **Zoltan_Struct**, which is now a copy of *from*.

C:
int **Zoltan_Copy_To** (
Zoltan_Struct **to*,
Zoltan_Struct **from*);

FORTRAN:
FUNCTION **Zoltan_Copy_To**(*to*, *from*)
INTEGER(Zoltan_INT) :: Zoltan_Copy_To
TYPE(Zoltan_Struct), INTENT(IN) :: to
TYPE(Zoltan_Struct), INTENT(IN) :: from

C++:
Zoltan & operator= (
const Zoltan &*zz*);

The **Zoltan_Copy_To** function copies one **Zoltan_Struct** to another, after first freeing any memory used by the target **Zoltan_Struct** and re-initializing it.

The C++ interface to the **Zoltan_Copy_To** function is through the **Zoltan** copy operator, which invokes the C library **Zoltan_Copy_To** program.

Arguments:	
to	A pointer to an existing **Zoltan_Struct**, the target of the copy.
from	A pointer to an existing **Zoltan_Struct**, the source of the copy.
Returned Value:	
int	**0** on success and **1** on failure.

C:
int **Zoltan_Set_Param** (
struct **Zoltan_Struct** **zz*,
char **param_name*,
char **new_val*);

FORTRAN:
FUNCTION **Zoltan_Set_Param**(*zz, param_name, new_val*)
INTEGER(Zoltan_INT) :: Zoltan_Set_Param
TYPE(Zoltan_Struct), INTENT(IN) :: zz
CHARACTER(LEN=*), INTENT(IN) :: param_name, new_value

C++:
int **Zoltan::Set_Param** (
const std::string &**param_name**,
const std::string &**new_value**);

Zoltan_Set_Param is used to alter the value of one of the parameters used by Zoltan. All Zoltan parameters have reasonable default values, but this routine allows a user to provide alternative values if desired.

Arguments:

zz	Pointer to the Zoltan structure created by **Zoltan_Create**.
param_name	A string containing the name of the parameter to be altered. Note that the string is case-insensitive. Also, different Zoltan structures can have different parameter values.
new_val	A string containing the new value for the parameter. Example strings include "3.154", "True", "7" or anything appropriate for the parameter being set. As above, the string is case-insensitive.

Returned Value:

int	Error code.

C: int **Zoltan_Set_Param_Vec** (
  struct **Zoltan_Struct** **zz*,
  char **param_name*,
  char **new_val*,
  int *index*);

FORTRAN: FUNCTION **Zoltan_Set_Param_Vec**(*zz*, *param_name*, *new_val*, *index*)
INTEGER(Zoltan_INT) :: Zoltan_Set_Param_Vec
TYPE(Zoltan_Struct), INTENT(IN) :: zz
CHARACTER(LEN=*), INTENT(IN) :: param_name, new_value
INTEGER(Zoltan_INT), INTENT(IN) :: index

C++: int **Zoltan::Set_Param_Vec** (
  const std::string &*param_name*,
  const std::string &*new_val*,
  const int &*index*);

Zoltan_Set_Param_Vec is used to alter the value of a vector parameter in Zoltan. A vector parameter is a parameter that has one name but contains multiple values. These values are referenced by their indices, usually starting at 0. Each entry (component) may have a different value. This routine sets a single entry (component) of a vector parameter. If you want all entries (components) of a vector parameter to have the same value, set the parameter using Zoltan_Set_Param as if it were a scalar parameter. If one only sets the values of a subset of the indices for a vector parameter, the remaining entries will have the default value for that particular parameter.

Arguments:

zz	Pointer to the Zoltan structure created by **Zoltan_Create**.
param_name	A string containing the name of the parameter to be altered. Note that the string is case-insensitive. Also, different Zoltan structures can have different parameter values.

new_val	A string containing the new value for the parameter. Example strings include "3.154", "True", "7" or anything appropriate for the parameter being set. As above, the string is case-insensitive.
index	The index of the entry of the vector parameter to be set. The default in Zoltan is that the first entry in a vector has index 0 (C-style indexing).
Returned Value:	
int	Error code.

C:

int **Zoltan_Set_Fn** (
  struct **Zoltan_Struct** ***zz*,
  ZOLTAN_FN_TYPE *fn_type*,
  void (**fn_ptr*)(),
  void **data*);

FORTRAN:

FUNCTION **Zoltan_Set_Fn**(*zz*, *fn_type*, *fn_ptr*, *data*)
INTEGER(Zoltan_INT) :: Zoltan_Set_Fn
TYPE(Zoltan_Struct), INTENT(IN) :: zz
TYPE(ZOLTAN_FN_TYPE), INTENT(IN) :: fn_type
EXTERNAL :: fn_ptr
<*type-data*>, OPTIONAL :: data

<*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.

C++:

int **Zoltan::Set_Fn** (
  const **ZOLTAN_FN_TYPE** &*fn_type*,
  void (**fn_ptr*)(),
  void **data* = 0);

Zoltan_Set_Fn registers an application-supplied query function in the Zoltan structure. All types of query functions can be registered through calls to **Zoltan_Set_Fn**. To register functions while maintaining strict type-checking of the *fn_ptr* argument, use **Zoltan_Set_<*zoltan_fn_type*>_Fn**.

Arguments:	
zz	Pointer to the Zoltan structure created by **Zoltan_Create**.
fn_type	The type of function being registered; see Application-Registered Query Functions for possible function types.
fn_ptr	A pointer to the application-supplied query function being registered.
data	A pointer to user defined data that will be passed, as an argument, to the function pointed to by *fn_ptr*. In C it may be NULL. In Fortran it may be omitted.
Returned Value:	
int	Error code.

C:	int **Zoltan_Set_<*zoltan_fn_type*>_Fn** (struct **Zoltan_Struct** **zz*, <***zoltan_fn_type***> (**fn_ptr*)(), void **data*);
FORTRAN:	FUNCTION **Zoltan_Set_<*zoltan_fn_type*>_Fn**(*zz*, *fn_ptr*, *data*) INTEGER(Zoltan_INT) :: Zoltan_Set_<*zoltan_fn_type*>_Fn TYPE(Zoltan_Struct), INTENT(IN) :: zz EXTERNAL :: fn_ptr <*type-data*>, OPTIONAL :: data An interface block for *fn_ptr* is included in the FUNCTION definition so that strict type-checking of the registered query function can be done. <*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.
C++:	int **Zoltan::Set_<*zoltan_fn_type*>_Fn** (<***zoltan_fn_type***> (**fn_ptr*)(), void **data* = 0);

The interface functions **Zoltan_Set_<*zoltan_fn_type*>_Fn**, where **<*zoltan_fn_type*>** is one of the query function types, register specific types of application-supplied query functions in the Zoltan structure. One interface function exists for each type of query function. For example, **Zoltan_Set_Num_Geom_Fn** registers a query function of type **ZOLTAN_NUM_GEOM_FN**. Each query function has an associated **Zoltan_Set_<*zoltan_fn_type*>_Fn**. A complete list of these functions is included in *include/zoltan.h*.

Query functions can be registered using either **Zoltan_Set_Fn** or **Zoltan_Set_<*zoltan_fn_type*>_Fn**. **Zoltan_Set_<*zoltan_fn_type*>_Fn** provides strict type checking of the *fn_ptr* argument; the argument's type is specified for each **Zoltan_Set_<*zoltan_fn_type*>_Fn**. **Zoltan_Set_Fn** does not provide this strict type checking, as the pointer to the registered function is cast to a void pointer.

Arguments:

zz	Pointer to the Zoltan structure created by **Zoltan_Create**.
fn_ptr	A pointer to the application-supplied query function being registered. The type of the pointer matches <***zoltan_fn_type***> in the name **Zoltan_Set_<*zoltan_fn_type*>_Fn**.
data	A pointer to user defined data that will be passed, as an argument, to the function pointed to by *fn_ptr*. In C it may be NULL. In Fortran it may be omitted.

Returned Value:

int	Error code.

Example:

The interface function
int **Zoltan_Set_Geom_Fn**(struct **Zoltan_Struct** *zz, **ZOLTAN_GEOM_FN** (*fn_ptr)(),
void *data);
registers an **ZOLTAN_GEOM_FN** query function.

C:	void **Zoltan_Destroy** (struct **Zoltan_Struct** ***zz*);
FORTRAN:	SUBROUTINE **Zoltan_Destroy**(*zz*) TYPE(Zoltan_Struct), POINTER :: zz
C++:	~**Zoltan** ();

Zoltan_Destroy frees the memory associated with a Zoltan structure and sets the structure to NULL in C or nullifies the structure in Fortran. Note that **Zoltan_Destroy** does not deallocate the import and export arrays returned from Zoltan (e.g., the arrays returned from **Zoltan_LB_Partition**); these arrays can be deallocated through a separate call to **Zoltan_LB_Free_Part**.

There is no explicit **Destroy** method in the C++ interface. The **Zoltan** object is destroyed when the destructor executes.

As a side effect, **Zoltan_Destroy** (and the C++ **Zoltan** destructor) frees the MPI communicator that had been allocated for the structure. So it is important that the application does not call **MPI_Finalize** before it calls **Zoltan_Destroy** or before the destructor executes.

Arguments:

zz	A pointer to the address of the Zoltan structure, created by **Zoltan_Create**, to be destroyed.

[Table of Contents | Next: Load-Balancing Functions | Previous: Zoltan Interface Functions]

Load-Balancing Functions

The following functions are the load-balancing interface functions in the Zoltan library; their descriptions are included below.

For backward compatibility with previous versions of Zoltan, the following functions are also maintained. These functions are applicable only when the number of partitions to be generated is equal to the number of processors on which the partitions are computed. That is, these functions assume "partitions" and "processors" are synonymous.

Descriptions of algorithm-specific interface functions are included with the documentation of their associated algorithms. Algorithm-specific functions include:

C: int **Zoltan_LB_Partition** (
struct **Zoltan_Struct** *zz*,
int *changes*,
int *num_gid_entries*,
int *num_lid_entries*,
int *num_import*,
ZOLTAN_ID_PTR *import_global_ids*,
ZOLTAN_ID_PTR *import_local_ids*,
int ***import_procs*,
int ***import_to_part*,
int *num_export*,
ZOLTAN_ID_PTR *export_global_ids*,
ZOLTAN_ID_PTR *export_local_ids*,
int ***export_procs*,
int ***export_to_part*);

FORTRAN: FUNCTION **Zoltan_LB_Partition**(*zz*, *changes*, *num_gid_entries*, *num_lid_entries*, *num_import*, *import_global_ids*, *import_local_ids*, *import_procs*, *import_to_part*, *num_export*, *export_global_ids*, *export_local_ids*, *export_procs*, *export_to_part*)
INTEGER(Zoltan_INT) :: Zoltan_LB_Partition
TYPE(Zoltan_Struct), INTENT(IN) :: zz
LOGICAL, INTENT(OUT) :: changes
INTEGER(Zoltan_INT), INTENT(OUT) :: num_gid_entries, num_lid_entries
INTEGER(Zoltan_INT), INTENT(OUT) :: num_import, num_export
INTEGER(Zoltan_INT), POINTER, DIMENSION(:) :: import_global_ids, export_global_ids
INTEGER(Zoltan_INT), POINTER, DIMENSION(:) :: import_local_ids, export_local_ids
INTEGER(Zoltan_INT), POINTER, DIMENSION(:) :: import_procs, export_procs
INTEGER(Zoltan_INT), POINTER, DIMENSION(:) :: import_to_part, export_to_part

C++: int **Zoltan::LB_Partition** (
int &*changes*,
int &*num_gid_entries*,
int &*num_lid_entries*,
int &*num_import*,
ZOLTAN_ID_PTR &*import_global_ids*,
ZOLTAN_ID_PTR &*import_local_ids*,
int * &*import_procs*,
int * &*import_to_part*,
int &*num_export*,
ZOLTAN_ID_PTR &*export_global_ids*,
ZOLTAN_ID_PTR &*export_local_ids*,
int * &*export_procs*,
int * &*export_to_part*);

Zoltan_LB_Partition invokes the load-balancing routine specified by the *LB_METHOD* parameter. The number of partitions it generates is specified by the *NUM_GLOBAL_PARTITIONS* or *NUM_LOCAL_PARTITIONS* parameters. Results of the partitioning are returned in lists of objects to be imported into and exported from partitions on this processor. Objects are included in these lists if *either* their partition assignment or their processor assignment is changed by the new decomposition. If an application requests multiple partitions on a single processor, these lists may include objects whose partition assignment is changing, but whose processor assignment is unchanged.

Returned arrays are allocated in Zoltan; applications should not allocate these arrays before calling **Zoltan_LB_Partition**. The arrays are later freed through calls to **Zoltan_LB_Free_Part**.

Arguments:

zz	Pointer to the Zoltan structure, created by **Zoltan_Create**, to be used in this invocation of the load-balancing routine.

changes	Set to 1 or .TRUE. if the decomposition was changed by the load-balancing method; 0 or .FALSE. otherwise.
num_gid_entries	Upon return, the number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter NUM_GID_ENTRIES.
num_lid_entries	Upon return, the number of array entries used to describe a single local ID. This value is the maximum value over all processors of the parameter NUM_LID_ENTRIES.
num_import	Upon return, the number of objects that are newly assigned to this processor or to partitions on this processor (i.e., the number of objects being imported from different partitions to partitions on this processor). If the value returned is -1, no import information has been returned and all import arrays below are NULL. (The RETURN_LISTS parameter determines whether import lists are returned).
import_global_ids	Upon return, an array of *num_import* global IDs of objects to be imported to partitions on this processor. (size = *num_import* * *num_gid_entries*)
import_local_ids	Upon return, an array of *num_import* local IDs of objects to be imported to partitions on this processor. (size = *num_import* * *num_lid_entries*)
import_procs	Upon return, an array of size *num_import* listing the processor IDs of the processors that owned the imported objects in the previous decomposition (i.e., the source processors).
import_to_part	Upon return, an array of size *num_import* listing the partitions to which the imported objects are being imported.
num_export	Upon return, this value of this count and the following lists depends on the value of the RETURN_LISTS parameter: • It is the count of objects on this processor that are newly assigned to other processors or to other partitions on this processor, if RETURN_LISTS is "EXPORT" or "EXPORT AND IMPORT". • It is the count of all objects on this processor, if RETURN_LISTS is "PARTITION ASSIGNMENTS". • It is -1 if the value of RETURN_LISTS indicates that either no lists are to be returned, or only import lists are to be returned. If the value returned is -1, no export information has been returned and all export arrays below are NULL .
export_global_ids	Upon return, an array of *num_export* global IDs of objects to be exported from partitions on this processor (if RETURN_LISTS is equal to "EXPORT" or "EXPORT AND IMPORT"), or an array of *num_export* global IDs for every object on this processor (if RETURN_LISTS is equal to "PARTITION ASSIGNMENTS"), . (size = *num_export* * *num_gid_entries*)

export_local_ids	Upon return, an array of *num_export* local IDs associated with the global IDs returned in *export_global_ids* (size = *num_export* * *num_lid_entries*)
export_procs	Upon return, an array of size *num_export* listing the processor ID of the processor to which each object is now assigned (i.e., the destination processor). If RETURN_LISTS is equal to "PARTITION ASSIGNMENTS", this list includes all objects, otherwise it only includes the objects which are moving to a new partition and/or process.
export_to_part	Upon return, an array of size *num_export* listing the partitions to which the objects are assigned under the new partitioning.

Returned Value:

int	Error code.

C: int **Zoltan_LB_Set_Part_Sizes** (
struct **Zoltan_Struct** **zz*,
int *global_num*,
int *len*,
int **part_ids*,
int **wgt_idx*,
float **part_sizes*);

FORTRAN: function **Zoltan_LB_Set_Part_Sizes**(*zz*,*global_part*,*len*,*partids*,*wgtidx*,*partsizes*)
integer(Zoltan_INT) :: Zoltan_LB_Set_Part_Sizes
type(Zoltan_Struct) INTENT(IN) zz
integer(Zoltan_INT) INTENT(IN) global_part,len,partids(*),wgtidx(*)
real(Zoltan_FLOAT) INTENT(IN) partsizes(*)

C++: int **Zoltan::LB_Set_Part_Sizes** (
const int &*global_num*,
const int &*len*,
int **part_ids*,
int **wgt_idx*,
float **part_sizes*);

Zoltan_LB_Set_Part_Sizes is used to specify the desired partition sizes in Zoltan. By default, Zoltan assumes that all partitions should be of equal size. With **Zoltan_LB_Set_Part_Sizes**, one can specify the relative (not absolute) sizes of the partitions. For example, if two partitions are requested and the desired sizes are 1 and 2, that means that the first partition will be assigned approximately one third of the total load. If the sizes were instead given as 1/3 and 2/3, respectively, the result would be exactly the same. Note that if there are multiple weights per object, one can (must) specify the partition size for each weight dimension independently.

Arguments:

zz	Pointer to the Zoltan structure created by **Zoltan_Create**.

global_num	Set to 1 if global partition numbers are given, 0 otherwise (local partition numbers).
len	Length of the next three input arrays.
part_ids	Array of partition numbers, either global or local. (Partition numbers are integers starting from 0.)
vwgt_idx	Array of weight indices (between 0 and OBJ_WEIGHT_DIM-1). This array should contain all zeros when there is only one weight per object.
part_sizes	Relative values for partition sizes; *part_sizes[i]* is the desired relative size of the *vwgt_idx[i]*'th weight of partition *part_ids[i]*.

Returned Value:

int	Error code.

C: int **Zoltan_LB_Eval** (
  struct **Zoltan_Struct** **zz*,
  int *print_stats*,
  int **nobj*,
  float **obj_wgt*,
  int **ncuts*,
  float **cut_wgt*,
  int **nboundary*,
  int **nadj*);

FORTRAN: FUNCTION **Zoltan_LB_Eval**(*zz, print_stats, nobj, obj_wgt, ncuts, cut_wgt, nboundary, nadj*)
INTEGER(Zoltan_INT) :: Zoltan_LB_Eval
TYPE(Zoltan_Struct), INTENT(IN) :: zz
LOGICAL, INTENT(IN) :: print_stats
INTEGER(Zoltan_INT), INTENT(OUT), OPTIONAL :: nobj, ncuts, nboundary, nadj
REAL(Zoltan_FLOAT), INTENT(OUT), DIMENSION(vwgt_dim), OPTIONAL :: obj_wgt
REAL(Zoltan_FLOAT), INTENT(OUT), DIMENSION(ewgt_dim), OPTIONAL :: cut_wgt

C++: int **Zoltan::LB_Eval** (
  const int &*print_stats*,
  int **nobj*,
  float * const *obj_wgt*,
  int **ncuts*,
  float * const *cut_wgt*,
  int **nboundary*,
  int **nadj*);

Zoltan_LB_Eval evaluates the quality of a decomposition. Some quality metrics are available only if the graph query functions have been registered. **Zoltan_LB_Eval** may either print a summary of the results to *stdout* or return the results in the output parameters. *NOTE:* The interface to this function may change in future versions of Zoltan. Users are discouraged from relying on the output arguments from

Zoltan_LB_Eval.

Arguments:	
zz	Pointer to the Zoltan structure.
print_stats	If *print_stats*>0 (.TRUE. in Fortran), print a summary (max, min, and sum) of the quality metrics to *stdout.*
nobj	Upon return, the number of objects on this processor.
obj_wgt	Upon return, an array (of dimension OBJ_WEIGHT_DIM) containing the sum of object weights on this processor.
ncuts	Upon return, the number of (communication) edge cuts for this processor.
cut_wgt	Upon return, an array (of dimension EDGE_WEIGHT_DIM) of cut weights for this processor.
nboundary	Upon return, the number of boundary objects on this processor.
nadj	Upon return, the number of adjacent processors as defined by the communication graph.
Returned Value:	
int	Error code.
Query functions:	
Required:	ZOLTAN_NUM_OBJ_FN ZOLTAN_OBJ_LIST_FN or ZOLTAN_FIRST_OBJ_FN/ZOLTAN_NEXT_OBJ_FN
Optional:	**ZOLTAN_NUM_EDGES_MULTI_F N** or **ZOLTAN_NUM_EDGES_FN** **ZOLTAN_EDGE_LIST_MULTI_F N** or **ZOLTAN_EDGE_LIST_FN**

An output parameter is returned only if the input value of that parameter was not NULL. The rationale for this feature is that if one wishes just to print the evaluation results, one can simply set all (or some of) the output parameters to NULL in the function call. From Fortran, one may omit one or more of the optional output parameters.

Note that the sum of *ncuts* over all processors is actually twice the number of edges cut in the graph (because each edge is counted twice). The same principle holds for *cut_wgt.*

There are a few improvements in Zoltan_LB_Eval in Zoltan version 1.5 (or higher). First, the balance data are computed with respect to both processors and partitions (if applicable). Second, the desired partition sizes (as set by Zoltan_LB_Set_Partition_Sizes) are taken into account when computing the imbalance.

Known bug: If a partition is spread across several processors, the computed cut information (*ncuts* and *cut_wgt)* may be incorrect (too high).

C:	int **Zoltan_LB_Free_Part** (
	    ZOLTAN_ID_PTR **global_ids,*
	    ZOLTAN_ID_PTR **local_ids,*
	    int ***procs*,
	    int ***to_part*);

FORTRAN:	FUNCTION **Zoltan_LB_Free_Part**(*global_ids, local_ids, procs, to_part*) INTEGER(Zoltan_INT) :: Zoltan_LB_Free_Part INTEGER(Zoltan_INT), POINTER, DIMENSION(:) :: global_ids INTEGER(Zoltan_INT), POINTER, DIMENSION(:) :: local_ids INTEGER(Zoltan_INT), POINTER, DIMENSION(:) :: procs, to_part

C++:	int **Zoltan::LB_Free_Part** (
	    ZOLTAN_ID_PTR **global_ids,*
	    ZOLTAN_ID_PTR **local_ids,*
	    int ***procs*,
	    int ***to_part*);

Zoltan_LB_Free_Part frees the memory allocated by Zoltan to return the results of **Zoltan_LB_Partition** or **Zoltan_Invert_Lists**. Memory pointed to by the arguments is freed and the arguments are set to NULL in C and C++ or nullified in Fortran. NULL arguments may be passed to **Zoltan_LB_Free_Part**. Note that this function does not destroy the Zoltan data structure itself; it is deallocated through a call to **Zoltan_Destroy** in C and Fortran and by the object destructor in C++.

Arguments:	
global_ids	An array containing the global IDs of objects.
local_ids	An array containing the local IDs of objects.
procs	An array containing processor IDs.
to_part	An array containing partition numbers.
Returned Value:	
int	Error code.

C:	int **Zoltan_LB_Balance** (
	    struct **Zoltan_Struct** **zz*,
	    int **changes*,
	    int **num_gid_entries*,
	    int **num_lid_entries*,
	    int **num_import*,
	    ZOLTAN_ID_PTR **import_global_ids,*
	    ZOLTAN_ID_PTR **import_local_ids,*
	    int ***import_procs*,
	    int **num_export*,
	    ZOLTAN_ID_PTR **export_global_ids,*
	    ZOLTAN_ID_PTR **export_local_ids,*

int ***export_procs*);

FORTRAN: FUNCTION **Zoltan_LB_Balance**(*zz, changes, num_gid_entries, num_lid_entries, num_import, import_global_ids, import_local_ids, import_procs, num_export, export_global_ids, export_local_ids, export_procs*)
INTEGER(Zoltan_INT) :: Zoltan_LB_Balance
TYPE(Zoltan_Struct), INTENT(IN) :: zz
LOGICAL, INTENT(OUT) :: changes
INTEGER(Zoltan_INT), INTENT(OUT) :: num_gid_entries, num_lid_entries
INTEGER(Zoltan_INT), INTENT(OUT) :: num_import, num_export
INTEGER(Zoltan_INT), POINTER, DIMENSION(:) :: import_global_ids, export_global_ids
INTEGER(Zoltan_INT), POINTER, DIMENSION(:) :: import_local_ids, export_local_ids
INTEGER(Zoltan_INT), POINTER, DIMENSION(:) :: import_procs, export_procs

---

Zoltan_LB_Balance is a wrapper around **Zoltan_LB_Partition** that excludes the partition assignment results. **Zoltan_LB_Balance** assumes the number of partitions is equal to the number of processors; thus, the partition assignment is equivalent to the processor assignment. Results of the partitioning are returned in lists of objects to be imported and exported. These arrays are allocated in Zoltan; applications should not allocate these arrays before calling **Zoltan_LB_Balance**. The arrays are later freed through calls to **Zoltan_LB_Free_Data** or **Zoltan_LB_Free_Part**.

Arguments:

All arguments are analogous to those in **Zoltan_LB_Partition**.
Partition-assignment arguments *import_to_part* and *export_to_part* are not included, as processor and partitions numbers are considered to be the same in **Zoltan_LB_Balance**.

Returned Value:

int — Error code.

C: int **Zoltan_LB_Free_Data** (
ZOLTAN_ID_PTR **import_global_ids*,
ZOLTAN_ID_PTR **import_local_ids*,
int ***import_procs*,
ZOLTAN_ID_PTR **export_global_ids*,
ZOLTAN_ID_PTR **export_local_ids*,
int ***export_procs*);

FORTRAN: FUNCTION **Zoltan_LB_Free_Data**(*import_global_ids*, *import_local_ids*, *import_procs*, *export_global_ids*, *export_local_ids*, *export_procs*)
INTEGER(Zoltan_INT) :: Zoltan_LB_Free_Data
INTEGER(Zoltan_INT), POINTER, DIMENSION(:) :: import_global_ids, export_global_ids
INTEGER(Zoltan_INT), POINTER, DIMENSION(:) :: import_local_ids, export_local_ids
INTEGER(Zoltan_INT), POINTER, DIMENSION(:) :: import_procs, export_procs

---

Zoltan_LB_Free_Data frees the memory allocated by the Zoltan to return the results of **Zoltan_LB_Balance** or **Zoltan_Compute_Destinations**. Memory pointed to by the arguments is freed and the arguments are set to NULL in C or nullified in Fortran. NULL arguments may be passed to **Zoltan_LB_Free_Data**. Note that this function does not destroy the Zoltan data structure itself; it is deallocated through a call to **Zoltan_Destroy**.

Arguments:

import_global_ids	The array containing the global IDs of objects imported to this processor.
import_local_ids	The array containing the local IDs of objects imported to this processor.
import_procs	The array containing the processor IDs of the processors that owned the imported objects in the previous decomposition (i.e., the source processors).
export_global_ids	The array containing the global IDs of objects exported from this processor.
export_local_ids	The array containing the local IDs of objects exported from this processor.
export_procs	The array containing the processor IDs of processors that own the exported objects in the new decomposition (i.e., the destination processors).

Returned Value:

int	Error code.

[Table of Contents | Next: Functions for Augmenting a Decomposition | Previous: Initialization Functions]

Functions for Augmenting a Decomposition

The following functions support the addition of new items to an existing decomposition. Given a decomposition, they determine to which processor(s) a new item should be assigned. Currently, they work in conjunction with only the RCB, RIB, and HSFC algorithms.

Zoltan_LB_Point_PP_Assign
Zoltan_LB_Box_PP_Assign

For backward compatibility with previous versions of Zoltan, the following functions are also maintained. These functions are applicable only when the number of partitions to be generated is equal to the number of processors on which the partitions are computed. That is, these functions assume "partitions" and "processors" are synonymous.

Zoltan_LB_Point_Assign
Zoltan_LB_Box_Assign

C:	int **Zoltan_LB_Point_PP_Assign** (struct **Zoltan_Struct** * *zz*, double * *coords*, int * *proc*, int * *part*);
FORTRAN:	FUNCTION **Zoltan_LB_Point_PP_Assign**(*zz, coords, proc, part*) INTEGER(Zoltan_INT) :: Zoltan_LB_Point_PP_Assign TYPE(Zoltan_Struct), INTENT(IN) :: zz REAL(Zoltan_DOUBLE), DIMENSION(*), INTENT(IN) :: coords INTEGER(Zoltan_INT), INTENT(OUT) :: proc INTEGER(Zoltan_INT), INTENT(OUT) :: part
C++:	int **Zoltan::LB_Point_PP_Assign** (double * const *coords*, int & *proc*, int & *part*);

Zoltan_LB_Point_PP_Assign is used to determine to which processor and partition a new point should be assigned. It is applicable only to geometrically generated decompositions (RCB, RIB, and HSFC). If the parameter **KEEP_CUTS** is set to TRUE, then the sequence of cuts that define the decomposition is saved. Given a new geometric point, the processor and partition which own it can be determined.

Arguments:

zz	Pointer to the Zoltan structure created by **Zoltan_Create**.
coords	The *(x,y)* or *(x,y,z)* coordinates of the point being assigned.
proc	Upon return, the ID of the processor to which the point should belong.
part	Upon return, the ID of the partition to which the point should belong.

Returned Value:

int	Error code.

C:	int **Zoltan_LB_Box_PP_Assign** (struct **Zoltan_Struct** * *zz*, double *xmin*, double *ymin*, double *zmin*, double *xmax*, double *ymax*, double *zmax*, int **procs*, int **numprocs*, int **parts*, int **numparts*);
FORTRAN:	FUNCTION **Zoltan_LB_Box_PP_Assign**(*zz, xmin, ymin, zmin, xmax, ymax, zmax, procs, numprocs, parts, numparts*) INTEGER(Zoltan_INT) :: Zoltan_LB_Box_PP_Assign TYPE(Zoltan_Struct), INTENT(IN) :: zz REAL(Zoltan_DOUBLE), INTENT(IN) :: xmin, ymin, zmin, xmax, ymax, zmax INTEGER(Zoltan_INT), DIMENSION(*), INTENT(OUT) ::procs INTEGER(Zoltan_INT), INTENT(OUT) :: numprocs INTEGER(Zoltan_INT), DIMENSION(*), INTENT(OUT) ::parts INTEGER(Zoltan_INT), INTENT(OUT) :: numparts
C++:	int **Zoltan::LB_Box_PP_Assign** (const double & *xmin*, const double & *ymin*, const double & *zmin*, const double & *xmax*, const double & *ymax*, const double & *zmax*, int * const *procs*, int & *numprocs*, int * const *parts*, int & *numparts*);

In many settings, it is useful to know which processors and partitions might need to know about an extended geometric object. **Zoltan_LB_Box_PP_Assign** addresses this problem. Given a geometric decomposition of space (currently only RCB, RIB, and HSFC are supported), and given an axis-aligned box around the geometric object, **Zoltan_LB_Box_PP_Assign** determines which processors and partitions own geometry that intersects the box. To use this routine, the parameter **KEEP_CUTS** must be set to TRUE when the decomposition is generated. This parameter will cause the sequence of geometric cuts to be saved, which is necessary for **Zoltan_LB_Box_PP_Assign** to do its job.

Note that if the parameter **REDUCE_DIMENSIONS** was set to TRUE and the geometry was determined to be degenerate when decomposition was calculated, then the calculation was performed on transformed coordinates. This means that **Zoltan_LB_Box_PP_Assign** must transform the supplied bounding box accordingly. The transformed vertices are bounded again, and the partition intersections are calculated in the transformed space on this new bounding box. The impact of this is that **Zoltan_LB_Box_PP_Assign** may return partitions not actually intersecting the original bounding box, but it will not omit any partitions intersecting the original bounding box.

Arguments:

zz	Pointer to the Zoltan structure created by **Zoltan_Create**.
xmin, *ymin*, *zmin*	The coordinates of the lower extent of the bounding box around the object. If the geometry is two-dimensional, the *z* value is ignored.
xmax, *ymax*, *zmax*	The coordinates of the upper extent of the bounding box around the object. If the geometry is two-dimensional, the *z* value is ignored.
procs	The list of processors intersecting the box are returned starting at this address. Note that *it is the responsibility of the calling routine to ensure that there is sufficient space for the return list.*
numprocs	Upon return, this value contains the number of processors that intersect the box (i.e. the number of entries placed in the *procs* list).
parts	The list of partitions intersecting the box are returned starting at this address. Note that *it is the responsibility of the calling routine to ensure that there is sufficient space for the return list.*
numparts	Upon return, this value contains the number of partitions that intersect the box (i.e. the number of entries placed in the *parts* list).

Returned Value:

int	Error code.

C: int **Zoltan_LB_Point_Assign** (
  struct **Zoltan_Struct** * *zz*,
  double * *coords*,
  int * *proc*);

FORTRAN: FUNCTION **Zoltan_LB_Point_Assign**(*zz*, *coords*, *proc*)
INTEGER(Zoltan_INT) :: Zoltan_LB_Point_Assign
TYPE(Zoltan_Struct), INTENT(IN) :: zz
REAL(Zoltan_DOUBLE), DIMENSION(*), INTENT(IN) :: coords
INTEGER(Zoltan_INT), INTENT(OUT) :: proc

Zoltan_LB_Point_Assign is is a wrapper around **Zoltan_LB_Point_PP_Assign** that excludes the partition assignment results. **Zoltan_LB_Point_Assign** assumes the number of partitions is equal to the number of processors; thus, the partition assignment is equivalent to the processor assignment.

Arguments:

All arguments are analogous to those in **[Zoltan_LB_Point_PP_Assign]**. Partition-assignment argument *part* is not included, as processor and partitions numbers are considered to be the same in **Zoltan_LB_Point_Assign**.

Returned Value:

int — [Error code].

C:

int **Zoltan_LB_Box_Assign** (
  struct **Zoltan_Struct** * *zz*,
  double *xmin*,
  double *ymin*,
  double *zmin*,
  double *xmax*,
  double *ymax*,
  double *zmax*,
  int **procs*,
  int **numprocs*);

FORTRAN:

FUNCTION **Zoltan_LB_Box_Assign**(*zz*, *xmin*, *ymin*, *zmin*, *xmax*, *ymax*, *zmax*, *procs*, *numprocs*)
INTEGER(Zoltan_INT) :: Zoltan_LB_Box_Assign
TYPE(Zoltan_Struct), INTENT(IN) :: zz
REAL(Zoltan_DOUBLE), INTENT(IN) :: xmin, ymin, zmin, xmax, ymax, zmax
INTEGER(Zoltan_INT), DIMENSION(*), INTENT(OUT) ::procs
INTEGER(Zoltan_INT), INTENT(OUT) :: numprocs

Zoltan_LB_Box_Assign is a wrapper around **[Zoltan_LB_Box_PP_Assign]** that excludes the partition assignment results. **Zoltan_LB_Box_Assign** assumes the number of partitions is equal to the number of processors; thus, the partition assignment is equivalent to the processor assignment.

Arguments:

All arguments are analogous to those in **[Zoltan_LB_Box_PP_Assign]**. Partition-assignment arguments *parts* and *numparts* are not included, as processor and partitions numbers are considered to be the same in **Zoltan_LB_Box_Assign**.

Returned Value:

int — [Error code].

[Table of Contents | Next: Migration Functions | Previous: Load-Balancing Functions]

Migration Functions

Zoltan's migration functions transfer object data to the processors in a new decomposition. Data to be transferred is specified through the import/export lists returned by **Zoltan_LB_Partition** (or **Zoltan_LB_Balance**). Alternatively, users may specify their own import/export lists.

The migration functions can migrate objects based on their new partition assignments and/or their new processor assignments. Behavior is determined by the **MIGRATE_ONLY_PROC_CHANGES** parameter.

If requested, Zoltan can automatically transfer an application's data between processors to realize a new decomposition. This functionality will be performed as part of the call to **Zoltan_LB_Partition** (or **Zoltan_LB_Balance**) if the **AUTO_MIGRATE** parameter is set to TRUE (nonzero) via a call to **Zoltan_Set_Param**. This approach is effective for when the data to be moved is relatively simple. For more complicated data movement, the application can leave **AUTO_MIGRATE** FALSE and call **Zoltan_Migrate** (or **Zoltan_Help_Migrate**) itself. In either case, routines to pack and unpack object data must be provided by the application. See the Migration Examples for examples with and without auto-migration.

The following functions are the migration interface functions. Their detailed descriptions can be found below.

- **Zoltan_Invert_Lists**
- **Zoltan_Migrate**

The following functions are maintained for backward compatibility with previous versions of Zoltan. These functions are applicable only when the number of partitions to be generated is equal to the number of processors on which the partitions are computed. That is, these functions assume "partitions" and "processors" are synonymous.

- **Zoltan_Compute_Destinations**
- **Zoltan_Help_Migrate**

C: int **Zoltan_Invert_Lists** (
struct **Zoltan_Struct** **zz*,
int *num_known*,
ZOLTAN_ID_PTR *known_global_ids*,
ZOLTAN_ID_PTR *known_local_ids*,
int **known_procs*,
int **known_to_part*,
int **num_found*,
ZOLTAN_ID_PTR **found_global_ids*,
ZOLTAN_ID_PTR **found_local_ids*,
int ***found_procs*,
int ***found_to_part*);

FORTRAN: FUNCTION **Zoltan_Invert_Lists**(*zz*, *num_known*, *known_global_ids*, *known_local_ids*, *known_procs*, *known_to_part*, *num_found*, *found_global_ids*, *found_local_ids*, *found_procs*, *found_to_part*)
INTEGER(Zoltan_INT) :: Zoltan_Invert_Lists
TYPE(Zoltan_Struct),INTENT(IN) :: zz
INTEGER(Zoltan_INT), INTENT(IN) :: num_known
INTEGER(Zoltan_INT), INTENT(OUT) :: num_found
INTEGER(Zoltan_INT), POINTER, DIMENSION(:) :: known_global_ids, found_global_ids
INTEGER(Zoltan_INT), POINTER, DIMENSION(:) :: known_local_ids, found_local_ids
INTEGER(Zoltan_INT), POINTER, DIMENSION(:) :: known_procs, found_procs
INTEGER(Zoltan_INT), POINTER, DIMENSION(:) :: known_to_part, found_to_part

C++: int **Zoltan::Invert_Lists** (
const int & *num_known*,
ZOLTAN_ID_PTR const *known_global_ids*,
ZOLTAN_ID_PTR const *known_local_ids*,
int * const *known_procs*,
int * const *known_to_part*,
int &*num_found*,
ZOLTAN_ID_PTR &*found_global_ids*,
ZOLTAN_ID_PTR &*found_local_ids*,
int * &*found_procs*,
int * &*found_to_part*);

Zoltan_Invert_Lists computes inverse communication maps useful for migrating data. It can be used in two ways:

- Given a list of known off-processor objects to be received by a processor, compute a list of local objects to be sent by the processor to other processors; or
- Given a list of known local objects to be sent by a processor to other processors, compute a list of off-processor objects to be received by the processor.

For example, if each processor knows which objects it will import from other processors,

Zoltan_Invert_Lists computes the list of objects each processor needs to export to other processors. If, instead, each processor knows which objects it will export to other processors, **Zoltan_Invert_Lists** computes the list of objects each processor will import from other processors. The computed lists are allocated in Zoltan; they should not be allocated by the application before calling **Zoltan_Invert_Lists**. These lists can be freed through a call to **Zoltan_LB_Free_Part**.

Arguments:

zz	Pointer to the Zoltan structure, created by **Zoltan_Create**, to be used in this invocation of the migration routine.
num_known	The number of known objects to be received (sent) by this processor.
known_global_ids	An array of *num_known* global IDs of known objects to be received (sent) by this processor. (size = *num_known* * NUM_GID_ENTRIES)
known_local_ids	An array of *num_known* local IDs of known objects to be received (sent) by this processor. (size = *num_known* * NUM_LID_ENTRIES)
known_procs	An array of size *num_known* listing the processor IDs of the processors that the known objects will be received from (sent to).
known_to_part	An array of size *num_known* listing the partition numbers of the partitions that the known objects will be assigned to.
num_found	Upon return, the number of objects that must be sent to (received from) other processors.
found_global_ids	Upon return, an array of *num_found* global IDs of objects to be sent (received) by this processor. (size = *num_found* * NUM_GID_ENTRIES)
found_local_ids	Upon return, an array of *num_found* local IDs of objects to be sent (received) by this processor. (size = *num_found* * NUM_LID_ENTRIES)
found_procs	Upon return, an array of size *num_found* listing the processor IDs of processors that the found objects will be sent to (received from).
found_to_part	An array of size *num_found* listing the partition numbers of the partitions that the found objects will be assigned to.

Returned Value:

int	Error code.

Note that the number of global and local ID entries (NUM_GID_ENTRIES and NUM_LID_ENTRIES) should be set using Zoltan_Set_Param before calling **Zoltan_Invert_Lists**. All processors must have the same values for these two parameters.

C:

int **Zoltan_Migrate** (
struct **Zoltan_Struct** *_zz_,
int _num_import_,
ZOLTAN_ID_PTR _import_global_ids_,
ZOLTAN_ID_PTR _import_local_ids_,
int *_import_procs_,
int *_import_to_part_,
int _num_export_,
ZOLTAN_ID_PTR _export_global_ids_,
ZOLTAN_ID_PTR _export_local_ids_,
int *_export_procs_,
int *_export_to_part_);

FORTRAN:

FUNCTION **Zoltan_Migrate**(_zz, num_import, import_global_ids, import_local_ids, import_procs, import_to_part, num_export, export_global_ids, export_local_ids, export_procs, export_to_part_)
INTEGER(Zoltan_INT) :: Zoltan_Migrate
TYPE(Zoltan_Struct),INTENT(IN) :: zz
INTEGER(Zoltan_INT), INTENT(IN) :: num_import, num_export
INTEGER(Zoltan_INT), POINTER, DIMENSION(:) :: import_global_ids, export_global_ids
INTEGER(Zoltan_INT), POINTER, DIMENSION(:) :: import_local_ids, export_local_ids
INTEGER(Zoltan_INT), POINTER, DIMENSION(:) :: import_procs, export_procs
INTEGER(Zoltan_INT), POINTER, DIMENSION(:) :: import_to_part, export_to_part

C++:

int **Zoltan::Migrate** (
const int & _num_import_,
ZOLTAN_ID_PTR const _import_global_ids_,
ZOLTAN_ID_PTR const _import_local_ids_,
int * const _import_procs_,
int * const _import_to_part_,
const int & _num_export_,
ZOLTAN_ID_PTR const _export_global_ids_,
ZOLTAN_ID_PTR const _export_local_ids_,
int * const _export_procs_,
int * const _export_to_part_);

Zoltan_Migrate takes lists of objects to be sent to other processors, along with the destinations of those objects, and performs the operations necessary to send the data associated with those objects to their destinations. **Zoltan_Migrate** performs the following operations using the application-registered functions:

- Call **ZOLTAN_PRE_MIGRATE_PP_FN_TYPE** (if registered)
- For each export object, call **ZOLTAN_OBJ_SIZE_FN_TYPE** to get object sizes.
- For each export object, call **ZOLTAN_PACK_OBJ_FN_TYPE** to load communication buffers.

- Communicate buffers to destination processors.
- Call **ZOLTAN_MID_MIGRATE_PP_FN_TYPE** (if registered).
- For each imported object, call **ZOLTAN_UNPACK_OBJ_FN_TYPE** to move data from the buffer into the new processor's data structures.
- Call **ZOLTAN_POST_MIGRATE_PP_FN_TYPE** (if registered).

Either export lists or import lists must be specified for **Zoltan_Migrate**. Both export lists and import lists may be specified, but both are not required.

If export lists are provided, non-NULL values for input arguments *import_global_ids*, *import_local_ids*, *import_procs*, and *import_to_part* are optional. The values must be non-NULL only if no export lists are provided or if the import lists are used by the application callback functions **ZOLTAN_PRE_MIGRATE_PP_FN**, **ZOLTAN_MID_MIGRATE_PP_FN**, and **ZOLTAN_POST_MIGRATE_PP_FN**. If all processors pass NULL arguments for the import arrays, the value of *num_import* should be -1.

Similarly, if import lists are provided, non-NULL values for input arguments *export_global_ids*, *export_local_ids*, *export_procs*, and *export_to_part* are optional. The values must be non-NULL only if no import lists are provided or if the export lists are used by the application callback functions **ZOLTAN_PRE_MIGRATE_PP_FN**, **ZOLTAN_MID_MIGRATE_PP_FN**, and **ZOLTAN_POST_MIGRATE_PP_FN**. If all processors pass NULL arguments for the export arrays, the value of *num_export* should be -1. In this case, **Zoltan_Migrate** computes the export lists based on the import lists.

Arguments:

zz	Pointer to the Zoltan structure, created by **Zoltan_Create**, to be used in this invocation of the migration routine.
num_import	The number of objects to be imported to partitions on this processor; these objects may be stored on other processors or may be moving to new partitions within this processor. Use *num_import*=-1 if all processors do not specify import arrays.
import_global_ids	An array of *num_import* global IDs of objects to be imported to partitions on this processor. (size = *num_import* * NUM_GID_ENTRIES). All processors may pass *import_global_ids*=NULL if export lists are provided and *import_global_ids* is not needed by callback functions **ZOLTAN_PRE_MIGRATE_PP_FN**, **ZOLTAN_MID_MIGRATE_PP_FN**, and **ZOLTAN_POST_MIGRATE_PP_FN**.

import_local_ids	An array of *num_import* local IDs of objects to be imported to partitions on this processor. (size = *num_import* * NUM_LID_ENTRIES) All processors may pass *import_local_ids*=NULL if export lists are provided and *import_local_ids* is not needed by callback functions **ZOLTAN_PRE_MIGRATE_PP_FN**, **ZOLTAN_MID_MIGRATE_PP_FN**, and **ZOLTAN_POST_MIGRATE_PP_FN**.
import_procs	An array of size *num_import* listing the processor IDs of objects to be imported to partitions on this processor (i.e., the source processors). All processors may pass *import_procs*=NULL if export lists are provided and *import_procs* is not needed by callback functions **ZOLTAN_PRE_MIGRATE_PP_FN**, **ZOLTAN_MID_MIGRATE_PP_FN**, and **ZOLTAN_POST_MIGRATE_PP_FN**.
import_to_part	An array of size *num_import* listing the partitions to which imported objects should be assigned. All processors may pass *import_to_part*=NULL if export lists are provided and *import_to_part* is not needed by callback functions **ZOLTAN_PRE_MIGRATE_PP_FN**, **ZOLTAN_MID_MIGRATE_PP_FN**, and **ZOLTAN_POST_MIGRATE_PP_FN**.
num_export	The number of objects that were stored on this processor in the previous decomposition that are assigned to other processors or to different partitions within this processor in the new decomposition. Use *num_export*=-1 if all processors do not specify export arrays.
export_global_ids	An array of *num_export* global IDs of objects to be exported to new partitions. (size = *num_export* * NUM_GID_ENTRIES) All processors may pass *export_global_ids*=NULL if import lists are provided and *export_global_ids* is not needed by callback functions **ZOLTAN_PRE_MIGRATE_PP_FN**, **ZOLTAN_MID_MIGRATE_PP_FN**, and **ZOLTAN_POST_MIGRATE_PP_FN**.
export_local_ids	An array of *num_export* local IDs of objects to be exported to new partitions. (size = *num_export* * NUM_LID_ENTRIES) All processors may pass *export_local_ids*=NULL if import lists are provided and *export_local_ids* is not needed by callback functions **ZOLTAN_PRE_MIGRATE_PP_FN**, **ZOLTAN_MID_MIGRATE_PP_FN**, and **ZOLTAN_POST_MIGRATE_PP_FN**.
export_procs	An array of size *num_export* listing the processor IDs to which exported objects should be assigned (i.e., the destination processors). All processors may pass *export_procs*=NULL if import lists are provided and *export_procs* is not needed by callback functions **ZOLTAN_PRE_MIGRATE_PP_FN**, **ZOLTAN_MID_MIGRATE_PP_FN**, and **ZOLTAN_POST_MIGRATE_PP_FN**.

export_to_part	An array of size *num_export* listing the partitions to which exported objects should be assigned. All processors may pass *export_to_part*=NULL if import lists are provided and *export_to_part* is not needed by callback functions **ZOLTAN_PRE_MIGRATE_PP_FN**, **ZOLTAN_MID_MIGRATE_PP_FN**, and **ZOLTAN_POST_MIGRATE_PP_FN**.

Returned Value:

int	Error code.

Note that the number of global and local ID entries (NUM_GID_ENTRIES and NUM_LID_ENTRIES) should be set using Zoltan_Set_Param before calling **Zoltan_Migrate**. All processors must have the same values for these two parameters.

C: int **Zoltan_Compute_Destinations** (
struct **Zoltan_Struct** **zz*,
int *num_known*,
ZOLTAN_ID_PTR *known_global_ids*,
ZOLTAN_ID_PTR *known_local_ids*,
int **known_procs*,
int **num_found*,
ZOLTAN_ID_PTR **found_global_ids*,
ZOLTAN_ID_PTR **found_local_ids*,
int ***found_procs*);

FORTRAN: FUNCTION **Zoltan_Compute_Destinations**(*zz, num_known, known_global_ids, known_local_ids, known_procs, num_found, found_global_ids, found_local_ids, found_procs*)
INTEGER(Zoltan_INT) :: Zoltan_Compute_Destinations
TYPE(Zoltan_Struct),INTENT(IN) :: zz
INTEGER(Zoltan_INT), INTENT(IN) :: num_known
INTEGER(Zoltan_INT), INTENT(OUT) :: num_found
INTEGER(Zoltan_INT), POINTER, DIMENSION(:) :: known_global_ids, found_global_ids
INTEGER(Zoltan_INT), POINTER, DIMENSION(:) :: known_local_ids, found_local_ids
INTEGER(Zoltan_INT), POINTER, DIMENSION(:) :: known_procs, found_procs

---

Zoltan_Compute_Destinations is a wrapper around **Zoltan_Invert_Lists** that excludes partition assignment arrays. It is maintained for backward compatibility with previous versions of Zoltan.

Zoltan_Compute_Destinations assumes the number of partitions is equal to the number of processors. The computed lists are allocated in Zoltan; they should not be allocated by the application before calling **Zoltan_Compute_Destinations**. These lists can be freed through a call to **Zoltan_LB_Free_Data** or **Zoltan_LB_Free_Part**.

Arguments:

All arguments are analogous to those in **Zoltan_Invert_Lists**.

Partition-assignment arrays *known_to_part* and *found_to_part* are not included, as partition and processor numbers are assumed to be the same in **Zoltan_Compute_Destinations**.

Returned Value:

int Error code.

Note that the number of global and local ID entries (NUM_GID_ENTRIES and NUM_LID_ENTRIES) should be set using Zoltan_Set_Param before calling **Zoltan_Compute_Destinations**. All processors must have the same values for these two parameters.

C:

int **Zoltan_Help_Migrate** (
    struct **Zoltan_Struct** **zz*,
    int *num_import*,
    ZOLTAN_ID_PTR *import_global_ids*,
    ZOLTAN_ID_PTR *import_local_ids*,
    int **import_procs*,
    int *num_export*,
    ZOLTAN_ID_PTR *export_global_ids*,
    ZOLTAN_ID_PTR *export_local_ids*,
    int **export_procs*);

FORTRAN:

FUNCTION **Zoltan_Help_Migrate**(*zz, num_import, import_global_ids, import_local_ids, import_procs, num_export, export_global_ids, export_local_ids, export_procs*)
INTEGER(Zoltan_INT) :: Zoltan_Help_Migrate
TYPE(Zoltan_Struct),INTENT(IN) :: zz
INTEGER(Zoltan_INT), INTENT(IN) :: num_import, num_export
INTEGER(Zoltan_INT), POINTER, DIMENSION(:) :: import_global_ids, export_global_ids
INTEGER(Zoltan_INT), POINTER, DIMENSION(:) :: import_local_ids, export_local_ids
INTEGER(Zoltan_INT), POINTER, DIMENSION(:) :: import_procs, export_procs

---

Zoltan_Help_Migrate is a wrapper around **Zoltan_Migrate** that excludes partition assignment arrays. It is maintained for backward compatibility with previous versions of Zoltan.

Zoltan_Help_Migrate assumes the number of partitions is equal to the number of processors. It uses migration pre-, mid-, and post-processing routines **ZOLTAN_PRE_MIGRATE_FN_TYPE**, **ZOLTAN_MID_MIGRATE_FN_TYPE**, and **ZOLTAN_POST_MIGRATE_FN_TYPE**, respectively, which also exclude partition assignment arrays.

Arguments:

All arguments are analogous to those in **Zoltan_Migrate**. Partition-assignment arrays *import_to_part* and *export_to_part* are not included, as partition and processor numbers are assumed to be the same in **Zoltan_Help_Migrate**.

Returned Value:

int Error code.

[Table of Contents | Next: Ordering Interface | Previous: Functions for Augmenting a Decomposition]

Ordering Functions

Zoltan provides limited capability for ordering a set of objects, typically given as a graph. The following functions are the ordering interface functions in the Zoltan library; their descriptions are included below.

[**Zoltan_Order**](#)

C:	int **Zoltan_Order** (struct **Zoltan_Struct** **zz*, int **num_gid_entries*, int **num_lid_entries*, int *num_obj*, **ZOLTAN_ID_PTR** *global_ids*, **ZOLTAN_ID_PTR** *local_ids*, int **rank*, int **iperm*, struct **Zoltan_Order_Struct** **order_info*);
FORTRAN:	FUNCTION **Zoltan_Order**(*zz*, *num_gid_entries*, *num_lid_entries*, *num_obj*, *global_ids*, *local_ids*, *rank*, *iperm*) INTEGER(Zoltan_INT) :: Zoltan_Order TYPE(Zoltan_Struct), INTENT(IN) :: zz INTEGER(Zoltan_INT), INTENT(OUT) :: num_gid_entries, num_lid_entries INTEGER(Zoltan_INT), INTENT(IN) :: num_obj INTEGER(Zoltan_INT) :: global_ids(*), local_ids(*) INTEGER(Zoltan_INT) :: rank(*), iperm(*)
C++:	int **Zoltan::Order** (int &*num_gid_entries*, int &*num_lid_entries*, const int &*num_obj*, **ZOLTAN_ID_PTR** *global_ids*, **ZOLTAN_ID_PTR** *local_ids*, int **rank*, int **iperm*);

Zoltan_Order invokes the ordering routine specified by the *ORDER_METHOD* parameter. Results of the ordering are returned in the arrays *rank* and *iperm*. *rank[i]* gives the rank of *global_ids[i]* in the computed ordering, while *iperm* is the inverse permutation of *rank*, that is, *iperm[rank[i]] = i.* The ordering may be either global or local, depending on *ORDER_TYPE*. The arrays *global_ids, local_ids, rank,* and *iperm* should all be allocated by the application before **Zoltan_Order** is called. Each array must have space for (at least) *num_obj* elements, where *num_obj* is the number of objects residing on a processor.

Arguments:

zz	Pointer to the Zoltan structure, created by **Zoltan_Create**, to be used in this invocation of the load-balancing routine.
num_gid_entries	Upon return, the number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter NUM_GID_ENTRIES.
num_lid_entries	Upon return, the number of array entries used to describe a single local ID. This value is the maximum value over all processors of the parameter NUM_LID_ENTRIES.
num_obj	Number of objects to order on this processor. At present, *num_obj* should be the total number of objects residing on a processor. In future releases, ordering only a subset of the objects may be permitted.
global_ids	An array of global IDs of objects to be ordered on this processor. (size = *num_obj* * *num_gid_entries*) The array may be uninitialized on input (if *REORDER* is false), but memory must have been allocated before **Zoltan_Order** is called.
local_ids	[Optional.] An array of local IDs of objects to be ordered on this processor. (size = *num_obj* * *num_lid_entries*) The array may be uninitialized on input (if *REORDER* is false), but memory must have been allocated before **Zoltan_Order** is called.
rank	Upon return, an array of length *num_obj* containing the rank of each object in the computed ordering. When rank[i] = j, that means that the object corresponding to *global_ids[i]* is the *j*th object in the ordering. (This array corresponds directly to the *perm* array in METIS and the *order* array in ParMETIS.) Note that the rank may refer to either a local or a global ordering, depending on ORDER_TYPE. Memory for this array must have been allocated before **Zoltan_Order** is called.
iperm	Upon return, an array of length *num_obj* containing the inverse permutation of *rank*. That is, *iperm[rank[i]] = i*. In other words, *iperm[j]* gives the *j*th object in the ordering. Memory for this array must have been allocated before **Zoltan_Order** is called.
order_info	Upon return, this struct contains additional information about the ordering produced. This parameter is currently not used and should always be set to NULL. It is not included in the FORTRAN or C++ interface.

Returned Value:

int	Error code.

Coloring Functions

Zoltan provides limited capability for coloring a set of objects, typically given as a graph. In graph coloring, each vertex is assigned an integer label such that no two adjacent vertices have the same label. The following functions are the coloring interface functions in the Zoltan library; their descriptions are included below.

Zoltan_Color

C: int **Zoltan_Color** (
    struct **Zoltan_Struct** **zz*,
    int **num_gid_entries*,
    int **num_lid_entries*,
    int *num_obj*,
    ZOLTAN_ID_PTR *global_ids*,
    ZOLTAN_ID_PTR *local_ids*,
    int **color_exp*);

Zoltan_Color invokes the coloring routine and the assigned colors of each object are returned in the array *color_exp*. *color_exp[i]*gives the color of *global_ids[i]* in the computed coloring. The arrays *global_ids*, *local_ids*,and *color_exp* should all be allocated by the application before **Zoltan_Color** is called. Each array must have space for (at least) *num_obj* elements, where *num_obj* is the number of objects residing on a processor.

Arguments:

zz	Pointer to the Zoltan structure, created by **Zoltan_Create**, to be used in this invocation of the load-balancing routine.
num_gid_entries	Upon return, the number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter NUM_GID_ENTRIES.
num_lid_entries	Upon return, the number of array entries used to describe a single local ID. This value is the maximum value over all processors of the parameter NUM_LID_ENTRIES.
num_obj	Number of objects to color on this processor. *num_obj* should be the total number of objects residing on a processor.
global_ids	Upon return, an array of global IDs of objects to be colored on this processor. (size = *num_obj* * *num_gid_entries*) Memory for this array must have been allocated before **Zoltan_Color** is called.
local_ids	[Optional.] Upon return, an array of local IDs of objects to be colored on this processor. (size = *num_obj* * *num_lid_entries*) Memory for this array must have been allocated before **Zoltan_Color** is called.

color_exp	Upon return, an array of length *num_obj* containing the colors of objects. That is, *color_exp[i]* gives the color of *global_ids[i]* in the computed coloring. (Colors are usually positive integers.) Memory for this array must have been allocated before **Zoltan_Color** is called.

Returned Value:

int	Error code.

[Table of Contents | Next: Application-Registered Query Functions | Previous: Ordering Functions]

Application-Registered Query Functions

Zoltan gets information about a processor's objects through calls to query functions. These functions must be provided by the application. They are "registered" with Zoltan; that is, a pointer to the function is passed to Zoltan, which can then call that function when its information is needed. Two categories of query functions are used by the library:

General Zoltan Query Functions
Migration Query Functions

In each category, a variety of query functions can be registered by the user. The query functions have a function type, describing their purpose. Functions can be registered with a Zoltan structure in two ways: through calls to **Zoltan_Set_Fn** or through calls to query-function-specific functions **Zoltan_Set_<*zoltan_fn_type*>_Fn**. When a function is registered through a call to **Zoltan_Set_Fn**, its function type is passed in the *fn_type* argument. When **Zoltan_Set_<*zoltan_fn_type*>_Fn** is used to register functions, the type of the function is implicit in the *fn_ptr* argument. Each function description below includes both its function type and function prototype.

Query functions that return information about data objects owned by a processor come in two forms: list-based functions that return information about a list of objects, and iterator functions that return information about a single object. Users can provide either version of the query function; they need not provide both. Zoltan calls the list-based functions with the IDs of all objects needed; this approach often provides faster performance as it eliminates the overhead of multiple function calls. List-based functions have the word "MULTI" in their function-type name. If, instead, the application provides iterator functions, Zoltan calls the iterator function once for each object whose data is needed. This approach, while slower, allows Zoltan to use less memory for some data.

Some algorithms in Zoltan require that certain query functions be registered by the application; for example, geometric partitioning algorithms such as Recursive Coordinate Bisection (RCB) require that either a **ZOLTAN_GEOM_FN** or a **ZOLTAN_GEOM_MULTI_FN** be registered. When a default value is specified below, the query function type is optional; if a function of that type is not registered, the default values are used. Details of which query functions are required by particular algorithms are included in the Algorithms section.

Many of the functions have both global and local object identifiers (IDs) in their argument lists. The global IDs provided by the application must be unique across all processors; they are used for identification within Zoltan. The local IDs are not used by Zoltan; they are provided for the convenience of the application and can be anything the application desires. The local IDs can be used by application query routines to enable direct access to application data. For example, the object with global ID "3295" may be stored by the application in location "15" of an array in the processor's local memory. Both global ID "3295" and local ID "15" can be used by the application to describe the object. Then, rather than searching the array for global ID "3295," the application query routines can subsequently use the local ID to index directly into the local storage array. See Data Types for Object IDs for a description of global and local IDs. All of the functions have, as their first argument, a pointer to data that is passed to

Zoltan through **Zoltan_Set_Fn** or **Zoltan_Set_<*zoltan_fn_type*>_Fn**. This data is not used by Zoltan. A different set of data can be supplied for each registered function. For example, if the local ID is an index into an array of data structures, then the data pointer might point to the head of the data structure array.

As their last argument, all functions have an error code that should be set and returned by the registered function.

If you are calling the Zoltan library from a C++ application, you may set the query function to be any class static function or any function defined outside of a class definition. However, it is possible you will wish to set the query function to be an object method. In that case, you should write a query function that takes a pointer to the object as it's *data* field. The query function can then call the object method.

[Table of Contents | Next: Load-Balancing Query Functions | Previous: Coloring Functions]

General Zoltan Query Functions

The following registered functions are used by various Zoltan algorithms in the Zoltan library. No single algorithm uses all the query functions; the algorithm descriptions indicate which query functions are required by individual algorithms.

Object ID Functions

C and C++:	typedef int **ZOLTAN_NUM_OBJ_FN** (void **data*, int **ierr*);
FORTRAN:	FUNCTION ***Get_Num_Obj***(*data*, *ierr*) INTEGER(Zoltan_INT) :: Get_Num_Obj <*type-data*>, INTENT(IN) :: data INTEGER(Zoltan_INT), INTENT(OUT) :: ierr <*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.

A **ZOLTAN_NUM_OBJ_FN** query function returns the number of objects that are currently assigned to the processor.

Function Type:	**ZOLTAN_NUM_OBJ_FN_TYPE**
Arguments:	
data	Pointer to user-defined data.
ierr	Error code to be set by function.
Returned Value:	
int	The number of objects that are assigned to the processor.

C and C++:	typedef void **ZOLTAN_OBJ_LIST_FN** (void **data*, int *num_gid_entries*, int *num_lid_entries*, **ZOLTAN_ID_PTR** *global_ids*, **ZOLTAN_ID_PTR** *local_ids*, int *wgt_dim*, float **obj_wgts*, int **ierr*);
FORTRAN:	SUBROUTINE ***Get_Obj_List***(*data, num_gid_entries, num_lid_entries, global_ids, local_ids, wgt_dim, obj_wgts, ierr*) <*type-data*>, INTENT(IN) :: data INTEGER(Zoltan_INT), INTENT(IN) :: num_gid_entries, num_lid_entries INTEGER(Zoltan_INT), INTENT(OUT), DIMENSION(*) :: global_ids INTEGER(Zoltan_INT), INTENT(OUT), DIMENSION(*) :: local_ids INTEGER(Zoltan_INT), INTENT(IN) :: wgt_dim REAL(Zoltan_FLOAT), INTENT(OUT), DIMENSION(*) :: obj_wgts INTEGER(Zoltan_INT), INTENT(OUT) :: ierr <*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.

A **ZOLTAN_OBJ_LIST_FN** query function fills two (three if weights are used) arrays with information about the objects currently assigned to the processor. Both arrays are allocated (and subsequently freed) by Zoltan; their size is determined by a call to a **ZOLTAN_NUM_OBJ_FN** query function to get the array size. For many algorithms, either a **ZOLTAN_OBJ_LIST_FN** query function or a **ZOLTAN_FIRST_OBJ_FN**/**ZOLTAN_NEXT_OBJ_FN** query-function pair must be registered; however, both query options need not be provided.

Function Type:	**ZOLTAN_OBJ_LIST_FN_TYPE**
Arguments:	
data	Pointer to user-defined data.
num_gid_entries	The number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter NUM_GID_ENTRIES.
num_lid_entries	The number of array entries used to describe a single local ID. This value is the maximum value over all processors of the parameter NUM_LID_ENTRIES.
global_ids	Upon return, an array of unique global IDs for all objects assigned to the processor.
local_ids	Upon return, an array of local IDs, the meaning of which can be determined by the application, for all objects assigned to the processor.
wgt_dim	The number of weights associated with an object (typically 1), or 0 if weights are not requested. This value is set through the parameter OBJ_WEIGHT_DIM.
obj_wgts	Upon return, an array of object weights. Weights for object *i* are stored in *obj_wgts[(i-1)*wgt_dim:i*wgt_dim-1]*. If *wgt_dim*=0, the return value of *obj_wgts* is undefined and may be NULL.
ierr	Error code to be set by function.

C and C++:	typedef int **ZOLTAN_FIRST_OBJ_FN** (void **data*, int *num_gid_entries*, int *num_lid_entries*, **ZOLTAN_ID_PTR** *first_global_id*, **ZOLTAN_ID_PTR** *first_local_id*, int *wgt_dim*, float **first_obj_wgt*, int **ierr*);

FORTRAN: FUNCTION ***Get_First_Obj***(*data, num_gid_entries, num_lid_entries, first_global_id, first_local_id, wgt_dim, first_obj_wgt, ierr*)
INTEGER(Zoltan_INT) :: Get_First_Obj
<*type-data*>, INTENT(IN) :: data
INTEGER(Zoltan_INT), INTENT(IN) :: num_gid_entries, num_lid_entries
INTEGER(Zoltan_INT), INTENT(OUT), DIMENSION(*) :: first_global_id
INTEGER(Zoltan_INT), INTENT(OUT), DIMENSION(*) :: first_local_id
INTEGER(Zoltan_INT), INTENT(IN) :: wgt_dim
REAL(Zoltan_FLOAT), INTENT(OUT), DIMENSION(*) :: first_obj_wgt
INTEGER(Zoltan_INT), INTENT(OUT) :: ierr

<*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.

A **ZOLTAN_FIRST_OBJ_FN** query function initializes an iteration over objects assigned to the processor. It returns the global and local IDs of the first object on the processor. Subsequent calls to a **ZOLTAN_NEXT_OBJ_FN** query function iterate over and return other objects assigned to the processor. This query-function pair frees the application from having to build an array of objects (as in **ZOLTAN_OBJ_LIST_FN**) and allows Zoltan's routines to obtain only as much information about objects as they need. For many algorithms, either a **ZOLTAN_OBJ_LIST_FN** query function or a **ZOLTAN_FIRST_OBJ_FN**/**ZOLTAN_NEXT_OBJ_FN** query-function pair must be registered; however, both query options need not be provided.

Function Type:	**ZOLTAN_FIRST_OBJ_FN_TYPE**
Arguments:	
data	Pointer to user-defined data.
num_gid_entries	The number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter NUM_GID_ENTRIES.
num_lid_entries	The number of array entries used to describe a single local ID. This value is the maximum value over all processors of the parameter NUM_LID_ENTRIES.
first_global_id	The returned value of the global ID for the first object; the value is ignored if there are no objects.
first_local_id	The returned value of the local ID for the first object; the value is ignored if there are no objects.
wgt_dim	The number of weights associated with an object (typically 1), or 0 if weights are not requested. This value is set through the parameter OBJ_WEIGHT_DIM.
first_obj_wgt	Upon return, the first object's weights; an array of length *wgt_dim.* Undefined if *wgt_dim=0.*
ierr	Error code to be set by function.
Returned Value:	
1	If *first_global_id* and *first_local_id* contain valid IDs of the first object.
0	If no objects are available.

C and C++: typedef int **ZOLTAN_NEXT_OBJ_FN** (void * *data*, int *num_gid_entries*, int *num_lid_entries*, **ZOLTAN_ID_PTR** *global_id*, **ZOLTAN_ID_PTR** *local_id*, **ZOLTAN_ID_PTR** *next_global_id*, **ZOLTAN_ID_PTR** *next_local_id*, int *wgt_dim*, float **next_obj_wgt*, int **ierr*);

FORTRAN: FUNCTION ***Get_Next_Obj***(*data, num_gid_entries, num_lid_entries, global_id, local_id, next_global_id, next_local_id, wgt_dim, next_obj_wgt, ierr*)
INTEGER(Zoltan_INT) :: Get_Next_Obj
<*type-data*>, INTENT(IN) :: data
INTEGER(Zoltan_INT), INTENT(IN) :: num_gid_entries, num_lid_entries
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: global_id
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: local_id
INTEGER(Zoltan_INT), INTENT(OUT), DIMENSION(*) :: next_global_id
INTEGER(Zoltan_INT), INTENT(OUT), DIMENSION(*) :: next_local_id
INTEGER(Zoltan_INT), INTENT(IN) :: wgt_dim
REAL(Zoltan_FLOAT), INTENT(OUT), DIMENSION(*) :: next_obj_wgt
INTEGER(Zoltan_INT), INTENT(OUT) :: ierr

<*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.

A **ZOLTAN_NEXT_OBJ_FN** query function is an iterator function which, when given an object assigned to the processor, returns the next object assigned to the processor. The first object of the iteration is provided by a **ZOLTAN_FIRST_OBJ_FN** query function. This query-function pair frees the application from having to build an array of objects (as in **ZOLTAN_OBJ_LIST_FN**) and allows Zoltan's routines to obtain only as much information about objects as they need. For many algorithms, either a **ZOLTAN_OBJ_LIST_FN** query function or a **ZOLTAN_FIRST_OBJ_FN**/**ZOLTAN_NEXT_OBJ_FN** query-function pair must be registered; however, both query options need not be provided.

Function Type:	**ZOLTAN_NEXT_OBJ_FN_TYPE**
Arguments:	
data	Pointer to user-defined data.
num_gid_entries	The number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter NUM_GID_ENTRIES.
num_lid_entries	The number of array entries used to describe a single local ID. This value is the maximum value over all processors of the parameter NUM_LID_ENTRIES.
global_id	The global ID of the previous object.
local_id	The local ID of the previous object.
next_global_id	The returned value of the global ID for the next object; the value is ignored if there are no more objects.
next_local_id	The returned value of the local ID for the next object; the value is ignored if there are no more objects.
wgt_dim	The number of weights associated with an object (typically 1), or 0 if weights are not requested. This value is set through the parameter OBJ_WEIGHT_DIM.
next_obj_wgt	Upon return, the next object's weights; an array of length *wgt_dim.* Undefined if *wgt_dim=0.*
ierr	Error code to be set by function.
Returned Value:	
1	If *next_global_id* and *next_local_id* contain valid IDs of the next object.
0	If no more objects are available.

C and C++: typedef void **ZOLTAN_PARTITION_MULTI_FN** (void **data*, int *num_gid_entries*, int *num_lid_entries*, int *num_obj*, **ZOLTAN_ID_PTR** *global_ids*, **ZOLTAN_ID_PTR** *local_ids*, int **parts*, int **ierr*);

FORTRAN: SUBROUTINE ***Get_Partition_Multi***(*data, num_gid_entries, num_lid_entries, num_obj, global_ids, local_ids, ierr*)
<*type-data*>, INTENT(IN) :: data
INTEGER(Zoltan_INT), INTENT(IN) :: num_gid_entries, num_lid_entries, num_obj
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: global_ids
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: local_ids
INTEGER(Zoltan_INT), INTENT(OUT), DIMENSION(*) :: parts
INTEGER(Zoltan_INT), INTENT(OUT) :: ierr

<*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.

A **ZOLTAN_PARTITION_MULTI_FN** query function returns a list of partitions to which given objects are currently assigned. If a **ZOLTAN_PARTITION_MULTI_FN** or **ZOLTAN_PARTITION_FN** is not registered, Zoltan assumes the partition numbers are the processor number of the owning processor. Valid partition numbers are non-negative integers.

Function Type:	**ZOLTAN_PARTITION_MULTI_FN_TYPE**
Arguments:	
data	Pointer to user-defined data.

num_gid_entries	The number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter NUM_GID_ENTRIES.
num_lid_entries	The number of array entries used to describe a single local ID. This value is the maximum value over all processors of the parameter NUM_LID_ENTRIES.
num_obj	The number of object IDs in arrays *global_ids* and *local_ids*.
global_ids	The global IDs of the objects for which the partition numbers should be returned.
local_ids	The local IDs of the objects for which the partition numbers should be returned.
parts	Upon return, an array of partition numbers corresponding to the global and local IDs.
ierr	Error code to be set by function.

C and C++: typedef int **ZOLTAN_PARTITION_FN** (void **data*, int *num_gid_entries*, int *num_lid_entries*, **ZOLTAN_ID_PTR** *global_id*, **ZOLTAN_ID_PTR** *local_id*, int **ierr*);

FORTRAN: FUNCTION ***Get_Partition***(*data*, *num_gid_entries*, *num_lid_entries*, *global_id*, *local_id*, *ierr*)

```
INTEGER(Zoltan_INT) :: Get_Partition
<type-data>, INTENT(IN) :: data
INTEGER(Zoltan_INT), INTENT(IN) :: num_gid_entries, num_lid_entries
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: global_id
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: local_id
INTEGER(Zoltan_INT), INTENT(OUT) :: ierr
```

<*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.

A **ZOLTAN_PARTITION_FN** query function returns the partition to which a given object is currently assigned. If a **ZOLTAN_PARTITION_FN** or **ZOLTAN_PARTITION_MULTI_FN** is not registered, Zoltan assumes the partition numbers are the processor number of the owning processor. Valid partition numbers are non-negative integers.

Function Type:	**ZOLTAN_PARTITION_FN_TYPE**
Arguments:	
data	Pointer to user-defined data.
num_gid_entries	The number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter NUM_GID_ENTRIES.
num_lid_entries	The number of array entries used to describe a single local ID. This value is the maximum value over all processors of the parameter NUM_LID_ENTRIES.
global_id	The global ID of the object for which the partition number should be returned.
local_id	The local ID of the object for which the partition number should be returned.
ierr	Error code to be set by function.
Returned Value:	
int	The partition number for the object identified by *global_id* and *local_id*.

Geometry-based Functions

C and C++: typedef int **ZOLTAN_NUM_GEOM_FN** (void **data*, int **ierr*);

FORTRAN:	FUNCTION ***Get_Num_Geom***(*data, ierr*) INTEGER(Zoltan_INT) :: Get_Num_Geom <*type-data*>, INTENT(IN) :: data INTEGER(Zoltan_INT), INTENT(OUT) :: ierr <*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on [Fortran query functions] for an explanation.

A **ZOLTAN_NUM_GEOM_FN** query function returns the number of values needed to express the geometry of an object. For example, for a two-dimensional mesh-based application, (*x*,*y*) coordinates are needed to describe an object's geometry; thus the **ZOLTAN_NUM_GEOM_FN** query function should return the value of two. For a similar three-dimensional application, the return value should be three.

Function Type:	**ZOLTAN_NUM_GEOM_FN_TYPE**
Arguments:	
data	Pointer to user-defined data.
ierr	Error code to be set by function.
Returned Value:	
int	The number of values needed to express the geometry of an object.

C and C++:	typedef void **ZOLTAN_GEOM_MULTI_FN** (void **data*, int *num_gid_entries*, int *num_lid_entries*, int *num_obj*, **ZOLTAN_ID_PTR** *global_ids*, **ZOLTAN_ID_PTR** *local_ids*, int *num_dim*, double **geom_vec*, int **ierr*);
FORTRAN:	SUBROUTINE ***Get_Geom_Multi***(*data, num_gid_entries, num_lid_entries, num_obj, global_ids, local_ids, num_dim, geom_vec, ierr*) <*type-data*>, INTENT(IN) :: data INTEGER(Zoltan_INT), INTENT(IN) :: num_gid_entries, num_lid_entries INTEGER(Zoltan_INT), INTENT(IN) :: num_obj, num_dim INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: global_ids INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: local_ids REAL(Zoltan_DOUBLE), INTENT(OUT), DIMENSION(*) :: geom_vec INTEGER(Zoltan_INT), INTENT(OUT) :: ierr <*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on [Fortran query functions] for an explanation.

A **ZOLTAN_GEOM_MULTI FN** query function returns a vector of geometry values for a list of given objects. The geometry vector is allocated by Zoltan to be of size *num_obj* * *num_dim*; its format is described below.

Function Type:	**ZOLTAN_GEOM_MULTI_FN_TYPE**
Arguments:	
data	Pointer to user-defined data.
num_gid_entries	The number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter [NUM_GID_ENTRIES].
num_lid_entries	The number of array entries used to describe a single local ID. This value is the maximum value over all processors of the parameter [NUM_LID_ENTRIES].
num_obj	The number of object IDs in arrays *global_ids* and *local_ids*.
global_ids	Array of global IDs of objects whose geometry values should be returned.
local_ids	Array of local IDs of objects whose geometry values should be returned.
num_dim	Number of coordinate entries per object (typically 1, 2, or 3).
geom_vec	Upon return, an array containing geometry values. For object *i* (specified by *global_ids[i*num_gid_entries]* and *local_ids[i*num_lid_entries]*, *i=0,1,...,num_obj-1*), coordinate values should be stored in *geom_vec[i*num_dim:(i+1)*num_dim-1]*.
ierr	Error code to be set by function.

C and C++:	typedef void **ZOLTAN_GEOM_FN** (void **data*, int *num_gid_entries*, int *num_lid_entries*, **ZOLTAN_ID_PTR** *global_id*, **ZOLTAN_ID_PTR** *local_id*, double **geom_vec*, int **ierr*);
FORTRAN:	SUBROUTINE ***Get_Geom***(*data, num_gid_entries, num_lid_entries, global_id,* *local_id, geom_vec, ierr*) <*type-data*>, INTENT(IN) :: data INTEGER(Zoltan_INT), INTENT(IN) :: num_gid_entries, num_lid_entries INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: global_id INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: local_id REAL(Zoltan_DOUBLE), INTENT(OUT), DIMENSION(*) :: geom_vec INTEGER(Zoltan_INT), INTENT(OUT) :: ierr <*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.

A **ZOLTAN_GEOM_FN** query function returns a vector of geometry values for a given object. The geometry vector is allocated by Zoltan to be of the size returned by a **ZOLTAN_NUM_GEOM_FN** query function.

Function Type:	**ZOLTAN_GEOM_FN_TYPE**
Arguments:	
data	Pointer to user-defined data.
num_gid_entries	The number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter NUM_GID_ENTRIES.
num_lid_entries	The number of array entries used to describe a single local ID. This value is the maximum value over all processors of the parameter NUM_LID_ENTRIES.
global_id	The global ID of the object whose geometry values should be returned.
local_id	The local ID of the object whose geometry values should be returned.
geom_vec	Upon return, an array containing geometry values.
ierr	Error code to be set by function.

Graph-based Functions

C and C++:	typedef void **ZOLTAN_NUM_EDGES_MULTI_FN** (void **data*, int *num_gid_entries*, int *num_lid_entries*, int *num_obj*, **ZOLTAN_ID_PTR** *global_ids*, **ZOLTAN_ID_PTR** *local_ids*, int **num_edges*, int **ierr*);
FORTRAN:	SUBROUTINE ***Get_Num_Edges_Multi***(*data, num_gid_entries, num_lid_entries,* *num_obj, global_ids, local_ids, num_edges, ierr*) INTEGER(Zoltan_INT) :: Get_Num_Edges <*type-data*>, INTENT(IN) :: data INTEGER(Zoltan_INT), INTENT(IN) :: num_gid_entries, num_lid_entries, num_obj INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: global_ids INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: local_ids INTEGER(Zoltan_INT), INTENT(OUT),DIMENSION(*) :: num_edges INTEGER(Zoltan_INT), INTENT(OUT) :: ierr <*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.

A **ZOLTAN_NUM_EDGES_MULTI_FN** query function returns the number of edges in the communication graph of the application for each object in a list of objects. That is, for each object in the *global_ids/local_ids* arrays, the number of objects with which the given object must share information is returned.

Function Type:	**ZOLTAN_NUM_EDGES_MULTI_FN_TYPE**
Arguments:	
data	Pointer to user-defined data.
num_gid_entries	The number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter NUM_GID_ENTRIES.
num_lid_entries	The number of array entries used to describe a single local ID. This value is the maximum value over all processors of the parameter NUM_LID_ENTRIES.
num_obj	The number of object IDs in arrays *global_ids* and *local_ids*.
global_ids	Array of global IDs of objects whose number of edges should be returned.
local_ids	Array of local IDs of objects whose number of edges should be returned.
num_edges	Upon return, an array containing numbers of edges. For object *i* (specified by *global_ids[i*num_gid_entries]* and *local_ids[i*num_lid_entries]*, *i=0,1,...,num_obj-1*), the number of edges should be stored in *num_edges[i]*.
ierr	Error code to be set by function.

C and C++: typedef int **ZOLTAN_NUM_EDGES_FN** (void **data*, int *num_gid_entries*, int *num_lid_entries*, **ZOLTAN_ID_PTR** *global_id*, **ZOLTAN_ID_PTR** *local_id*, int **ierr*);

FORTRAN: FUNCTION ***Get_Num_Edges***(*data*, *num_gid_entries*, *num_lid_entries*, *global_id*, *local_id*, *ierr*)
INTEGER(Zoltan_INT) :: Get_Num_Edges
<*type-data*>, INTENT(IN) :: data
INTEGER(Zoltan_INT), INTENT(IN) :: num_gid_entries, num_lid_entries
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: global_id
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: local_id
INTEGER(Zoltan_INT), INTENT(OUT) :: ierr

<*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.

A **ZOLTAN_NUM_EDGES_FN** query function returns the number of edges for a given object in the communication graph of the application (i.e., the number of objects with which the given object must share information).

Function Type:	**ZOLTAN_NUM_EDGES_FN_TYPE**
Arguments:	
data	Pointer to user-defined data.
num_gid_entries	The number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter NUM_GID_ENTRIES.
num_lid_entries	The number of array entries used to describe a single local ID. This value is the maximum value over all processors of the parameter NUM_LID_ENTRIES.
global_id	The global ID of the object for which the number of edges should be returned.
local_id	The local ID of the object for which the number of edges should be returned.
ierr	Error code to be set by function.
Returned Value:	
int	The number of edges for the object identified by *global_id* and *local_id*.

C and C++: typedef void **ZOLTAN_EDGE_LIST_MULTI_FN** (void **data*, int *num_gid_entries*, int *num_lid_entries*, int *num_obj*, **ZOLTAN_ID_PTR** *global_ids*, **ZOLTAN_ID_PTR** *local_ids*, int **num_edges*, **ZOLTAN_ID_PTR** *nbor_global_id*, int **nbor_procs*, int *wgt_dim*, float **ewgts*, int **ierr*);

FORTRAN:	SUBROUTINE ***Get_Edge_List_Multi***(*data, num_gid_entries, num_lid_entries, num_obj, global_ids, local_ids, num_edges, nbor_global_id, nbor_procs, wgt_dim, ewgts, ierr*) <*type-data*>, INTENT(IN) :: data INTEGER(Zoltan_INT), INTENT(IN) :: num_gid_entries, num_lid_entries, num_obj INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: global_ids INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: local_ids INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: num_edges INTEGER(Zoltan_INT), INTENT(OUT), DIMENSION(*) :: nbor_global_id INTEGER(Zoltan_INT), INTENT(OUT), DIMENSION(*) :: nbor_procs INTEGER(Zoltan_INT), INTENT(IN) :: wgt_dim REAL(Zoltan_FLOAT), INTENT(OUT), DIMENSION(*) :: ewgts INTEGER(Zoltan_INT), INTENT(OUT) :: ierr <*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.

A **ZOLTAN_EDGE_LIST_MULTI_FN** query function returns lists of global IDs, processor IDs, and optionally edge weights for objects sharing edges with objects specified in the *global_ids* input array; objects share edges when they must share information with other objects. The arrays for the returned neighbor lists are allocated by Zoltan; their size is determined by a calls to **ZOLTAN_NUM_EDGES_MULTI_FN** or **ZOLTAN_NUM_EDGES_FN** query functions.

Function Type:	**ZOLTAN_EDGE_LIST_MULTI_FN_TYPE**
Arguments:	
data	Pointer to user-defined data.
num_gid_entries	The number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter NUM_GID_ENTRIES.
num_lid_entries	The number of array entries used to describe a single local ID. This value is the maximum value over all processors of the parameter NUM_LID_ENTRIES.
num_obj	The number of object IDs in arrays *global_ids* and *local_ids*.
global_ids	Array of global IDs of objects whose edge lists should be returned.
local_ids	Array of local IDs of objects whose edge lists should be returned.
num_edges	An array containing numbers of edges for each object in *global_ids*. For object *i* (specified by *global_ids[i*num_gid_entries]* and *local_ids[i*num_lid_entries]*, *i=0,1,...,num_obj-1*), the number of edges is stored in *num_edges[i]*.
nbor_global_id	Upon return, an array of global IDs of objects sharing edges with the objects specified in *global_ids*. For object *i* (specified by *global_ids[i*num_gid_entries]* and *local_ids[i*num_lid_entries]*, *i=0,1,...,num_obj-1*), edges are stored in *nbor_global_id[sum*num_gid_entries]* to *nbor_global_id[(sum+num_edges[i])*num_gid_entries-1]*, where *sum* = the sum of *num_edges[j]* for *j=0,1,...,i-1*.
nbor_procs	Upon return, an array of processor IDs that identifies where the neighboring objects reside. For neighboring object *i* (stored in *nbor_global_id[i*num_gid_entries]*), the processor owning the neighbor is stored in *nbor_procs[i]*.
wgt_dim	The number of weights associated with an edge (typically 1), or 0 if edge weights are not requested. This value is set through the parameter EDGE_WEIGHT_DIM.
ewgts	Upon return, an array of edge weights, where *ewgts[i*wgt_dim:(i+1)*wgt_dim-1]* corresponds to the weights for the *i*th edge. If *wgt_dim=0*, the return value of *ewgts* is undefined and may be NULL.
ierr	Error code to be set by function.

C and C++:	typedef void **ZOLTAN_EDGE_LIST_FN** (void **data*, int *num_gid_entries*, int *num_lid_entries*, **ZOLTAN_ID_PTR** *global_id*, **ZOLTAN_ID_PTR** *local_id*, **ZOLTAN_ID_PTR** *nbor_global_id*, int **nbor_procs*, int *wgt_dim*, float **ewgts*, int **ierr*);

FORTRAN:	SUBROUTINE ***Get_Edge_List***(*data, num_gid_entries, num_lid_entries, global_id, local_id, nbor_global_id, nbor_procs, wgt_dim, ewgts, ierr*) <*type-data*>, INTENT(IN) :: data INTEGER(Zoltan_INT), INTENT(IN) :: num_gid_entries, num_lid_entries INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: global_id INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: local_id INTEGER(Zoltan_INT), INTENT(OUT), DIMENSION(*) :: nbor_global_id INTEGER(Zoltan_INT), INTENT(OUT), DIMENSION(*) :: nbor_procs INTEGER(Zoltan_INT), INTENT(IN) :: wgt_dim REAL(Zoltan_FLOAT), INTENT(OUT), DIMENSION(*) :: ewgts INTEGER(Zoltan_INT), INTENT(OUT) :: ierr <*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.

A **ZOLTAN_EDGE_LIST_FN** query function returns lists of global IDs, processor IDs, and optionally edge weights for objects sharing an edge with a given object (i.e., objects that must share information with the given object). The arrays for the returned neighbor lists are allocated by Zoltan; their size is determined by a call to **ZOLTAN_NUM_EDGES_MULTI_FN** or **ZOLTAN_NUM_EDGES_FN** query functions.

Function Type:	**ZOLTAN_EDGE_LIST_FN_TYPE**
Arguments:	
data	Pointer to user-defined data.
num_gid_entries	The number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter NUM_GID_ENTRIES.
num_lid_entries	The number of array entries used to describe a single local ID. This value is the maximum value over all processors of the parameter NUM_LID_ENTRIES.
global_id	The global ID of the object for which an edge list should be returned.
local_id	The local ID of the object for which an edge list should be returned.
nbor_global_id	Upon return, an array of global IDs of objects sharing edges with the given object.
nbor_procs	Upon return, an array of processor IDs that identifies where the neighboring objects reside.
wgt_dim	The number of weights associated with an edge (typically 1), or 0 if edge weights are not requested. This value is set through the parameter EDGE_WEIGHT_DIM.
ewgts	Upon return, an array of edge weights, where *ewgts[i*wgt_dim:(i+1)*wgt_dim-1]* corresponds to the weights for the *i*th edge. If *wgt_dim=0*, the return value of *ewgts* is undefined and may be NULL.
ierr	Error code to be set by function.

Hypergraph-based Functions

C and C++:	typedef void **ZOLTAN_HG_SIZE_CS_FN** (void **data*, int **num_lists*, int **num_pins*, int **format*, int **ierr*);
FORTRAN:	SUBROUTINE ***Get_HG_Size_CS***(*data, num_lists, num_pins, format, ierr*) <*type-data*>, INTENT(IN) :: data INTEGER(Zoltan_INT), INTENT(OUT) :: num_lists INTEGER(Zoltan_INT), INTENT(OUT) :: num_pins INTEGER(Zoltan_INT), INTENT(OUT) :: format INTEGER(Zoltan_INT), INTENT(OUT) :: ierr <*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.

A hypergraph (which may alternatively be viewed as a sparse matrix) can be supplied to the Zoltan library in one of two compressed storage formats. In compressed hyperedge format (ZOLTAN_COMPRESSED_EDGE) a list of global hyperedge IDs is provided. Then a single list of the hypergraph pins, is provided. A pin is the connection between a vertex and a hyperedge (corresponds to a nonzero in a sparse matrix). Pins do not have separate IDs but are rather identified by the global ID of the vertex containing the pin, and implicitly also by the hyperedge ID. An example is provided below.

The other format is compressed vertex (ZOLTAN_COMPRESSED_VERTEX). In this format a list of vertex global IDs is provided. Then a list of pins ordered by vertex and then by hyperedge is provided. The pin ID in this case is the global ID of the row (or hyperedge) in which the pin appears. In both formats, an array must be provided pointing to the start in the list of pins where each row or column begins. Sparse matrix users may think of these two formats as CSR (compressed sparse row) and CSC (compressed sparse column) format, respectively.

The point of this query function is to tell Zoltan in which format the application will supply the hypergraph, how many vertices and hyperedges there will be, and how many pins. The actual hypergraph is supplied with a query function of the type ZOLTAN_HG_CS_FN_TYPE.

This query function is required by all applications using the hypergraph methods of Zoltan (unless they are using the graph-based functions with hypergraph code instead).

Function Type:	**ZOLTAN_HG_SIZE_CS_FN_TYPE**
Arguments:	
data	Pointer to user-defined data.
num_lists	Upon return, the number of rows (if using compressed row storage) or columns (if using compressed column storage) that will be supplied to Zoltan by the application process.
num_pins	Upon return, the number of pins (matrix non-zeroes) that will be supplied to Zoltan by the application process.
format	Upon return, the format in which the application process will provide the hypergraph to Zoltan. The options are **ZOLTAN_COMPRESSED_EDGE** and **ZOLTAN_COMPRESSED_VERTEX**.
ierr	Error code to be set by function.

C and C++:	typedef void **ZOLTAN_HG_CS_FN** (void **data*, int *num_gid_entries*, int *num_row_or_col*, int *num_pins*, int *format*, **ZOLTAN_ID_PTR** *vtxedge_GID*, int **vtxedge_ptr*, **ZOLTAN_ID_PTR** *pin_GID*, int **ierr*);
FORTRAN:	SUBROUTINE ***Get_HG_CS***(*data*, *num_gid_entries*, *num_row_or_col*, *num_pins*, *format*, *vtxedge_GID*, *vtxedge_ptr*, *pin_GID*, *ierr*) <*type-data*>, INTENT(IN) :: data INTEGER(Zoltan_INT), INTENT(IN) :: num_gid_entries, num_row_or_col, num_pins, format INTEGER(Zoltan_INT), INTENT(OUT), DIMENSION(*) :: vtxedge_GID INTEGER(Zoltan_INT), INTENT(OUT), DIMENSION(*) :: vtxedge_ptr INTEGER(Zoltan_INT), INTENT(OUT), DIMENSION(*) :: pin_GID INTEGER(Zoltan_INT), INTENT(OUT) :: ierr <*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.

A ZOLTAN_HG_CS_FN returns a hypergraph, in a sparse matrix-like style. The size and format of the data to be returned must have been supplied to Zoltan using a ZOLTAN_HG_SIZE_CS_FN_TYPE function.

When a hypergraph is distributed across multiple processes, Zoltan expects that all processes share a consistent global numbering scheme for hyperedges and vertices. Also, no two processes should return the same pin (matrix non-zero) in this query function.

This query function is required by all applications using the hypergraph methods of Zoltan (unless they are using the graph-based functions with hypergraph code instead).

Function Type:	**ZOLTAN_HG_CS_FN_TYPE**
Arguments:	
data	Pointer to user-defined data.
num_gid_entries	The number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter NUM_GID_ENTRIES.
num_vtx_edge	The number of global IDs that is expected to appear on return in *vtxedge_GID*.
num_pins	The number of pins that is expected to appear on return in *pin_GID*.
format	If *format* is **ZOLTAN_COMPRESSED_EDGE**, Zoltan expects that row (hyperedge) global IDs will be returned in *vtxedge_GID*, and that column (vertex) global IDs will be returned in *pin_GIDs*. If it is **ZOLTAN_COMPRESSED_VERTEX**, then column global IDs are expected to be returned in *vtxedge_GID* and row global IDs are expected to be returned in *pin_GIDs*.
vtxedge_GID	Upon return, a list of *num_row_or_col* global IDs.
vtxedge_ptr	Upon return, this array contains *num_row_or_col* integers. The integer in the *i'th* array element is the index in array *pin_GID* where the pins for the *i'th* row (if format is **ZOLTAN_COMPRESSED_EDGE**) or *i'th* column (if format is **ZOLTAN_COMPRESSED_VERTEX**) begins. Array indices begin at zero.

pin_GID	Upon return, a list of *num_pins* global IDs. This is the list of the pins (or matrix non-zeros) contained in the rows or columns listed in *vtxedge_GID*.
ierr	Error code to be set by function.

Example

	vertex				
hyperedge	**10**	**20**	**30**	**40**	**50**
1	0	0	1	1	0
2	0	1	1	0	0
3	1	0	0	0	1

Compressed hyperedge storage:

vtxedge_GID = {1, 2, 3}
vtxedge_ptr = {0, 2, 4}
pin_GID = {30, 40, 20, 30, 10, 50}

Compressed vertex storage:

vtxedge_GID = {10, 20, 30, 40, 50}
vtxedge_ptr = {0, 1, 2, 4, 5}
pin_GID = {3, 2, 1, 2, 1, 3}

C and C++:	typedef void **ZOLTAN_HG_SIZE_EDGE_WTS_FN** (void **data*, int **num_edges*, int **ierr*);
FORTRAN:	SUBROUTINE ***Get_HG_Size_Edge_Wts***(*data, num_edges, ierr*) <*type-data*>, INTENT(IN) :: data INTEGER(Zoltan_INT), INTENT(OUT) :: num_edges INTEGER(Zoltan_INT), INTENT(OUT) :: ierr <*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.

A ZOLTAN_HG_SIZE_EDGE_WTS_FN returns the number of hyperedges for which a process will supply edge weights. The number of weights per hyperedge was supplied by the application with the EDGE_WEIGHT_DIM parameter. The actual edge weights will be supplied with a ZOLTAN_HG_EDGE_WTS_FN_TYPE function.

This query function is not required. If no hyperedge weights are supplied, Zoltan will assume every hyperedge has weight 1.0.

Function Type:	**ZOLTAN_HG_SIZE_EDGE_WTS_FN_TYPE**
Arguments:	
data	Pointer to user-defined data.
num_edges	Upon return, the number of hyperedges for which edge weights will be supplied.
ierr	Error code to be set by function.

C and C++:	typedef void **ZOLTAN_HG_EDGE_WTS_FN** (void **data*, int *num_gid_entries*, int *num_lid_entries*, int *num_edges*, int *edge_weight_dim*, **ZOLTAN_ID_PTR** *edge_GID*, **ZOLTAN_ID_PTR** *edge_LID*, float **edge_weight*, int **ierr*);
FORTRAN:	SUBROUTINE ***Get_HG_Edge_Wts***(*data*, *num_gid_entries*, *num_lid_entries*, *num_edges*, *edge_weight_dim*, *edge_GID*, *edge_LID*, *edge_weight*, *ierr*) <*type-data*>, INTENT(IN) :: data INTEGER(Zoltan_INT), INTENT(IN) :: num_gid_entries, num_lid_entries, num_edges, edge_weight_dim INTEGER(Zoltan_INT), INTENT(OUT), DIMENSION(*) :: edge_GID INTEGER(Zoltan_INT), INTENT(OUT), DIMENSION(*) :: edge_LID REAL(Zoltan_FLOAT), INTENT(OUT), DIMENSION(*) :: edge_weight INTEGER(Zoltan_INT), INTENT(OUT) :: ierr <*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.

A ZOLTAN_HG_EDGE_WTS_FN returns edges weights for a set of hypergraph edges. The number of weights supplied for each hyperedge should equal the value of the EDGE_WEIGHT_DIM parameter. In the case of a hypergraph which is distributed across multiple processes, if more than one process supplies edge weights for the same hyperedge, the different edge weights will be resolved according to the value of the PHG_EDGE_WEIGHT_OPERATION parameter.

This query function is not required. If no hyperedge weights are supplied, Zoltan will assume every hyperedge has weight 1.0.

Function Type:	**ZOLTAN_HG_SIZE_EDGE_WTS_FN_TYPE**
Arguments:	
data	Pointer to user-defined data.
num_gid_entries	The number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter NUM_GID_ENTRIES.
num_lid_entries	The number of array entries used to describe a single local ID. This value is the maximum value over all processors of the parameter NUM_LID_ENTRIES.
num_edges	The number of hyperedges for which edge weights should be supplied in the *edge_weight* array.
edge_weight_dim	The number of weights which should be supplied for each hyperedge. This is also the value of the EDGE_WEIGHT_DIM parameter.
edge_GID	Upon return, this array should contain the global IDs of the *num_edges* hyperedges for which the application is supplying edge weights.
edge_LID	Upon return, this array can optionally contain the local IDs of the *num_edges* hyperedges for which the application is supplying edge weights.
edge_weight	Upon return, this array should contain the weights for each edge listed in the *edge_GID*. If *edge_weight_dim* is greater than one, all weights for one hyperedge are listed before the weights for the next hyperedge are listed.
ierr	Error code to be set by function.

Tree-based Functions

C and C++: typedef int **ZOLTAN_NUM_COARSE_OBJ_FN** (void **data*, int **ierr*);

FORTRAN: FUNCTION ***Get_Num_Coarse_Obj***(*data*, *ierr*)
INTEGER(Zoltan_INT) :: Get_Num_Coarse_Obj
<*type-data*>, INTENT(IN) :: data
INTEGER(Zoltan_INT), INTENT(OUT) :: ierr

<*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.

A ZOLTAN_NUM_COARSE_OBJ_FN query function returns the number of objects (elements) in the initial coarse grid.

Function Type: **ZOLTAN_NUM_COARSE_OBJ_FN_TYPE**
Arguments:
data Pointer to user-defined data.
ierr Error code to be set by function.
Returned Value:
int The number of objects in the coarse grid.

C and C++: typedef void **ZOLTAN_COARSE_OBJ_LIST_FN** (void **data*, int *num_gid_entries*, int *num_lid_entries*, **ZOLTAN_ID_PTR** *global_ids*, **ZOLTAN_ID_PTR** *local_ids*, int **assigned*, int **num_vert*, **ZOLTAN_ID_PTR** *vertices*, int **in_order*, **ZOLTAN_ID_PTR** *in_vertex*, **ZOLTAN_ID_PTR** *out_vertex*, int **ierr*);

FORTRAN: SUBROUTINE ***Get_Coarse_Obj_List***(*data*, *num_gid_entries*, *num_lid_entries*, *global_ids*, *local_ids*, *assigned*, *num_vert*, *vertices*, *in_order*, *in_vertex*, *out_vertex*, *ierr*)
<*type-data*>, INTENT(IN) :: data
INTEGER(Zoltan_INT), INTENT(IN) :: num_gid_entries, num_lid_entries
INTEGER(Zoltan_INT), INTENT(OUT), DIMENSION(*) :: global_ids
INTEGER(Zoltan_INT), INTENT(OUT), DIMENSION(*) :: local_ids
INTEGER(Zoltan_INT), INTENT(OUT), DIMENSION(*) :: assigned, num_vert, vertices, in_vertex, out_vertex
INTEGER(Zoltan_INT), INTENT(OUT) :: in_order, ierr

<*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.

A ZOLTAN_COARSE_OBJ_LIST_FN query function returns lists of global IDs, local IDs, vertices, and order information for all objects (elements) of the initial coarse grid. The vertices are designated by a global ID such that if two elements share a vertex then the same ID

designates that vertex in both elements and on all processors. The user may choose to provide the order in which the elements should be traversed or have Zoltan determine the order. If the user provides the order, then entry and exit vertices for a path through the elements may also be provided. The arrays for the returned values are allocated by Zoltan; their size is determined by a call to a ZOLTAN_NUM_COARSE_OBJ_FN query function.

Function Type:	**ZOLTAN_COARSE_OBJ_LIST_FN_TYPE**
Arguments:	
data	Pointer to user-defined data.
num_gid_entries	The number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter NUM_GID_ENTRIES.
num_lid_entries	The number of array entries used to describe a single local ID. This value is the maximum value over all processors of the parameter NUM_LID_ENTRIES.
global_ids	Upon return, an array of global IDs of all objects in the coarse grid.
local_ids	Upon return, an array of local IDs of all objects in the coarse grid.
assigned	Upon return, an array of integers indicating whether or not each object is currently assigned to this processor. A value of 1 indicates it is assigned to this processor; a value of 0 indicates it is assigned to some other processor. For elements that have been refined, it is ignored unless weights are assigned to interior nodes of the tree.
num_vert	Upon return, an array containing the number of vertices for each object.
vertices	Upon return, an array of global IDs of the vertices of each object. If the number of vertices for objects 0 through *i*-1 is *N*, then the vertices for object *i* are in *vertices*[*N**num_gid_entries: (N+num_vert*[*i*]*)*num_gid_entries*]
in_order	Upon return, 1 if the user is providing the objects in the order in which they should be traversed, or 0 if Zoltan should determine the order.
in_vertex	Upon return, an array of global IDs of the vertices through which to enter each element in the user provided traversal. It is required only if the user is providing the order for the coarse grid objects (i.e., *in_order*==1) and allowing Zoltan to select the order of the children in at least one invocation of **ZOLTAN_CHILD_LIST_FN**.
out_vertex	Upon return, an array of global IDs of the vertex through which to exit each element in the user provided traversal. The same provisions hold as for *in_vertex*.
ierr	Error code to be set by function.

C and C++:	typedef int **ZOLTAN_FIRST_COARSE_OBJ_FN** (void **data*, int *num_gid_entries*, int *num_lid_entries*, **ZOLTAN_ID_PTR** *global_id*, **ZOLTAN_ID_PTR** *local_id*, int **assigned*, int **num_vert*, **ZOLTAN_ID_PTR** *vertices*, int **in_order*, **ZOLTAN_ID_PTR** *in_vertex*, **ZOLTAN_ID_PTR** *out_vertex*, int **ierr*);
FORTRAN:	FUNCTION ***Get_First_Coarse_Obj***(*data, num_gid_entries, num_lid_entries, global_id, local_id, assigned, num_vert, vertices, in_order, in_vertex, out_vertex, ierr*) INTEGER(Zoltan_INT) :: Get_First_Coarse_Obj <*type-data*>, INTENT(IN) :: data INTEGER(Zoltan_INT), INTENT(IN) :: num_gid_entries, num_lid_entries INTEGER(Zoltan_INT), INTENT(OUT), DIMENSION(*) :: global_id INTEGER(Zoltan_INT), INTENT(OUT), DIMENSION(*) :: local_id INTEGER(Zoltan_INT), INTENT(OUT) :: assigned, num_vert, in_order, ierr INTEGER(Zoltan_INT), INTENT(OUT), DIMENSION(*) :: vertices, in_vertex, out_vertex <*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section

on Fortran query functions for an explanation.

A ZOLTAN_FIRST_COARSE_OBJ_FN query function initializes an iteration over the objects of the initial coarse grid. It returns the global ID, local ID, vertices, and order information for the first object (element) of the initial coarse grid. Subsequent calls to a ZOLTAN_NEXT_COARSE_OBJ_FN iterate over and return other objects from the coarse grid. The vertices are designated by a global ID such that if two elements share a vertex then the same ID designates that vertex in both elements and on all processors. The user may choose to provide the order in which the elements should be traversed, or have Zoltan determine the order. If the user provides the order, then entry and exit vertices for a path through the elements may also be provided.

Function Type:	**ZOLTAN_FIRST_COARSE_OBJ_FN_TYPE**
Arguments:	
data	Pointer to user-defined data.
num_gid_entries	The number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter NUM_GID_ENTRIES.
num_lid_entries	The number of array entries used to describe a single local ID. This value is the maximum value over all processors of the parameter NUM_LID_ENTRIES.
global_ids	Upon return, the global ID of the first object in the coarse grid.
local_ids	Upon return, the local ID of the first object in the coarse grid.
assigned	Upon return, an integer indicating whether or not this object is currently assigned to this processor. A value of 1 indicates it is assigned to this processor; a value of 0 indicates it is assigned to some other processor. For elements that have been refined, it is ignored unless weights are assigned to interior nodes of the tree.
num_vert	Upon return, the number of vertices for this object.
vertices	Upon return, an array of global IDs of the vertices of this object.
in_order	Upon return, 1 if the user is providing the objects in the order in which they should be traversed, or 0 if Zoltan should determine the order.
in_vertex	Upon return, the vertex through which to enter this element in the user provided traversal. It is required only if the user is providing the order for the coarse grid objects (i.e., *in_order*==1) and allowing Zoltan to select the order of the children in at least one invocation of **ZOLTAN_CHILD_LIST_FN**.
out_vertex	Upon return, the vertex through which to exit this element in the user provided traversal. The same provisions hold as for *in_vertex*.
ierr	Error code to be set by function.
Returned Value:	
1	If *global_id* and *local_id* contain valid IDs of the first object in the coarse grid.
0	If no coarse grid is available.

C and C++: typedef int **ZOLTAN_NEXT_COARSE_OBJ_FN** (void **data*, int *num_gid_entries*, int *num_lid_entries*, **ZOLTAN_ID_PTR** *global_id*, **ZOLTAN_ID_PTR** *local_id*, **ZOLTAN_ID_PTR** *next_global_id*, **ZOLTAN_ID_PTR** *next_local_id*, int **assigned*, int **num_vert*, **ZOLTAN_ID_PTR** *vertices*, **ZOLTAN_ID_PTR** *in_vertex*, **ZOLTAN_ID_PTR** *out_vertex*, int **ierr*);

FORTRAN:	FUNCTION ***Get_Next_Coarse_Obj***(*data, num_gid_entries, num_lid_entries, global_id, local_id, next_global_id, next_local_id, assigned, num_vert, vertices, in_vertex, out_vertex, ierr*) INTEGER(Zoltan_INT) :: Get_Next_Coarse_Obj <*type-data*>, INTENT(IN) :: data INTEGER(Zoltan_INT), INTENT(IN) :: num_gid_entries, num_lid_entries INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: global_id INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: local_id INTEGER(Zoltan_INT), INTENT(OUT), DIMENSION(*) :: next_global_id INTEGER(Zoltan_INT), INTENT(OUT), DIMENSION(*) :: next_local_id INTEGER(Zoltan_INT), INTENT(OUT) :: assigned, num_vertex, ierr INTEGER(Zoltan_INT), INTENT(OUT), DIMENSION(*) :: vertices, in_vertex, out_vertex <*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.

A ZOLTAN_NEXT_COARSE_OBJ_FN query function is an iterator function that returns the next object in the initial coarse grid. The first object of the iteration is provided by a ZOLTAN_FIRST_COARSE_OBJ_FN query function.

Function Type:	**ZOLTAN_NEXT_COARSE_OBJ_FN_TYPE**
Arguments:	
data	Pointer to user-defined data.
num_gid_entries	The number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter NUM_GID_ENTRIES.
num_lid_entries	The number of array entries used to describe a single local ID. This value is the maximum value over all processors of the parameter NUM_LID_ENTRIES.
global_id	The global ID of the previous object in the coarse grid.
local_id	The local ID of the previous object in the coarse grid.
next_global_id	Upon return, the global ID of the next object in the coarse grid.
next_local_id	Upon return, the local ID of the next object in the coarse grid.
assigned	Upon return, an integer indicating whether or not this object is currently assigned to this processor. A value of 1 indicates it is assigned to this processor; a value of 0 indicates it is assigned to some other processor. For elements that have been refined, it is ignored unless weights are assigned to interior nodes of the tree.
num_vert	Upon return, the number of vertices for this object.
vertices	Upon return, an array of global IDs of the vertices of this object.
in_vertex	Upon return, the vertex through which to enter this element in the user provided traversal. It is required only if the user is providing the order for the coarse grid objects (i.e., *in_order*==1) and allowing Zoltan to select the order of the children in at least one invocation of **ZOLTAN_CHILD_LIST_FN**.
out_vertex	Upon return, the vertex through which to exit this element in the user provided traversal. The same provisions hold as for *in_vertex*.
ierr	Error code to be set by function.
Returned Value:	
1	If *global_id* and *local_id* contain valid IDs of the next object in the coarse grid.
0	If no more objects are available.

C and C++:	typedef int **ZOLTAN_NUM_CHILD_FN** (void **data*, int *num_gid_entries*, int *num_lid_entries*, **ZOLTAN_ID_PTR** *global_id*, **ZOLTAN_ID_PTR** *local_id*, int **ierr*);

FORTRAN:	FUNCTION ***Get_Num_Child***(*data, num_gid_entries, num_lid_entries, global_id, local_id, ierr*) INTEGER(Zoltan_INT) :: Get_Num_Child <*type-data*>, INTENT(IN) :: data INTEGER(Zoltan_INT), INTENT(IN) :: num_gid_entries, num_lid_entries INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: global_id INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: local_id INTEGER(Zoltan_INT), INTENT(OUT) :: ierr <*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.

A ZOLTAN_NUM_CHILD_FN query function returns the number of children of the element with the given global and local IDs. If the element has not been refined, the number of children is 0.

Function Type:	**ZOLTAN_NUM_CHILD_FN_TYPE**
Arguments:	
data	Pointer to user-defined data.
num_gid_entries	The number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter NUM_GID_ENTRIES.
num_lid_entries	The number of array entries used to describe a single local ID. This value is the maximum value over all processors of the parameter NUM_LID_ENTRIES.
global_id	The global ID of the object for which the number of children is requested.
local_id	The local ID of the object for which the number of children is requested.
ierr	Error code to be set by function.
Returned Value:	
int	The number of children.

C and C++:	typedef void **ZOLTAN_CHILD_LIST_FN** (void **data*, int *num_gid_entries*, int *num_lid_entries*, **ZOLTAN_ID_PTR** *parent_gid*, **ZOLTAN_ID_PTR** *parent_lid*, **ZOLTAN_ID_PTR** *child_gids*, **ZOLTAN_ID_PTR** *child_lids*, int **assigned*, int **num_vert*, **ZOLTAN_ID_PTR** *vertices*, **ZOLTAN_REF_TYPE** **ref_type*, **ZOLTAN_ID_PTR** *in_vertex*, **ZOLTAN_ID_PTR** *out_vertex*, int **ierr*);
FORTRAN:	SUBROUTINE ***Get_Child_List***(*data, num_gid_entries, num_lid_entries, parent_gid, parent_lid, child_gids, child_lids, assigned, num_vert, vertices, ref_type, in_vertex, out_vertex, ierr*) <*type-data*>, INTENT(IN) :: data INTEGER(Zoltan_INT), INTENT(IN) :: num_gid_entries, num_lid_entries INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: parent_gid INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: parent_lid INTEGER(Zoltan_INT), INTENT(OUT), DIMENSION(*) :: child_gids INTEGER(Zoltan_INT), INTENT(OUT), DIMENSION(*) :: child_lids INTEGER(Zoltan_INT), INTENT(OUT), DIMENSION(*) :: assigned, num_vert, vertices, in_vertex, out_vertex INTEGER(Zoltan_INT), INTENT(OUT) :: ref_type, ierr <*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.

A ZOLTAN_CHILD_LIST_FN query function returns lists of global IDs,

local IDs, vertices, and order information for all children of a refined element. The vertices are designated by a global ID such that if two elements share a vertex then the same ID designates that vertex in both elements and on all processors. The user may choose to provide the order in which the children should be traversed, or have Zoltan determine the order based on the type of element refinement used to create the children. If the user provides the order, then entry and exit vertices for a path through the elements may also be provided. The arrays for the returned values are allocated by Zoltan; their size is determined by a call to a ZOLTAN_NUM_CHILD_FN query function.

Function Type:	**ZOLTAN_CHILD_LIST_FN_TYPE**
Arguments:	
data	Pointer to user-defined data.
num_gid_entries	The number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter NUM_GID_ENTRIES.
num_lid_entries	The number of array entries used to describe a single local ID. This value is the maximum value over all processors of the parameter NUM_LID_ENTRIES.
parent_gid	The global ID of the object whose children are requested.
parent_lid	The local ID of the object whose children are requested.
child_gids	Upon return, an array of global IDs of all children of this object.
child_lids	Upon return, an array of local IDs of all children of this object.
assigned	Upon return, an array of integers indicating whether or not each child is currently assigned to this processor. A value of 1 indicates it is assigned to this processor; a value of 0 indicates it is assigned to some other processor. For children that have been further refined, it is ignored unless weights are assigned to interior nodes of the tree.
num_vert	Upon return, an array containing the number of vertices for each object.
vertices	Upon return, an array of global IDs of the vertices of each object. If the number of vertices for objects 0 through *i*-1 is *N*, then the vertices for object *i* are in *vertices*[*N*num_gid_entries: (N+num_vert*[*i*]*)*num_gid_entries*]
ref_type	Upon return, a value indicating what type of refinement was used to create the children. This determines how the children will be ordered. The values currently supported are: *ZOLTAN_TRI_BISECT* Bisection of triangles. *ZOLTAN_QUAD_QUAD* Quadrasection of quadrilaterals. *ZOLTAN_HEX3D_OCT* Octasection of hexahedra. *ZOLTAN_OTHER_REF* All other forms of refinement. *ZOLTAN_IN_ORDER* Traverse the children in the order in which they are provided.
in_vertex	Upon return, an array of global IDs of the vertex through which to enter each element in the user provided traversal. It is required only if the user is providing the order for the children of this element (i.e., *ref_type==ZOLTAN_IN_ORDER*) but does not provide the order for the children of at least one of those children.
out_vertex	Upon return, an array of global IDs of the vertex through which to exit each element in the user provided traversal. The same provisions hold as for *in_vertex*.
ierr	Error code to be set by function.

C and C++:	typedef void **ZOLTAN_CHILD_WEIGHT_FN** (void **data*, int *num_gid_entries*, int *num_lid_entries*, **ZOLTAN_ID_PTR** *global_id*, **ZOLTAN_ID_PTR** *local_id*, int *wgt_dim*, float **obj_wgt*, int **ierr*);

FORTRAN: SUBROUTINE ***Get_Child_Weight***(*data, num_gid_entries, num_lid_entries, global_id, local_id, wgt_dim, obj_wgt, ierr*)
<*type-data*>, INTENT(IN) :: data
INTEGER(Zoltan_INT), INTENT(IN) :: num_gid_entries, num_lid_entries
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: global_id
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: local_id
INTEGER(Zoltan_INT), INTENT(IN) :: wgt_dim
REAL(Zoltan_FLOAT), INTENT(OUT), DIMENSION(*) :: obj_wgt
INTEGER(Zoltan_INT), INTENT(OUT) :: ierr

<*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.

A ZOLTAN_CHILD_WEIGHT_FN query function returns the weight of an object. Interior nodes of the refinement tree as well as the leaves are allowed to have weights.

Function Type: **ZOLTAN_CHILD_WEIGHT_FN_TYPE**

Arguments:

data	Pointer to user-defined data.
num_gid_entries	The number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter NUM_GID_ENTRIES.
num_lid_entries	The number of array entries used to describe a single local ID. This value is the maximum value over all processors of the parameter NUM_LID_ENTRIES.
global_id	The global ID of the object whose weight is requested.
local_id	The local ID of the object whose weight is requested.
wgt_dim	The number of weights associated with an object (typically 1), or 0 if weights are not requested. This value is set through the parameter OBJ_WEIGHT_DIM.
obj_wgt	Upon return, an array containing the object's weights. If *wgt_dim*=0, the return value of *obj_wgts* is undefined and may be NULL.
ierr	Error code to be set by function.

Border Object Functions (currently not used)

C: typedef int **ZOLTAN_NUM_BORDER_OBJ_FN** (void **data*, int *nbor_proc*, int **ierr*);

FORTRAN: FUNCTION ***Get_Num_Border_Obj***(*data, nbor_proc, ierr*)
INTEGER(Zoltan_INT) :: Get_Num_Border_Obj
<*type-data*>, INTENT(IN) :: data
INTEGER(Zoltan_INT), INTENT(IN) :: nbor_proc
INTEGER(Zoltan_INT), INTENT(OUT) :: ierr

<*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.

A ZOLTAN_NUM_BORDER_OBJ_FN query function returns the number of objects sharing a processor subdomain border (in the communication graph of the application) with a given processor.

Function Type:	**ZOLTAN_NUM_BORDER_OBJ_FN_TYPE**
Arguments:	
data	Pointer to user-defined data.
nbor_proc	The processor ID of the processor for which the number of border objects should be returned.
ierr	Error code to be set by function.
Returned Value:	
int	The number of objects sharing a processor subdomain border with processor *nbor_proc*.

C: typedef void **ZOLTAN_BORDER_OBJ_LIST_FN** (void **data*, int *num_gid_entries*, int *num_lid_entries*, int *nbor_proc*, **ZOLTAN_ID_PTR** *global_ids*, **ZOLTAN_ID_PTR** *local_ids*, int *wgt_dim*, float **obj_wgts*, int **ierr*);

FORTRAN: SUBROUTINE ***Get_Border_Obj_List***(*data, num_gid_entries, num_lid_entries, nbor_proc, global_ids, local_ids, wgt_dim, obj_wgts, ierr*)
<*type-data*>, INTENT(IN) :: data
INTEGER(Zoltan_INT), INTENT(IN) :: nbor_proc
INTEGER(Zoltan_INT), INTENT(IN) :: num_gid_entries, num_lid_entries
INTEGER(Zoltan_INT), INTENT(OUT), DIMENSION(*) :: global_ids
INTEGER(Zoltan_INT), INTENT(OUT), DIMENSION(*) :: local_ids
INTEGER(Zoltan_INT), INTENT(IN) :: wgt_dim
REAL(Zoltan_FLOAT), INTENT(OUT), DIMENSION(*) :: obj_wgts
INTEGER(Zoltan_INT), INTENT(OUT) :: ierr

<*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.

A ZOLTAN_BORDER_OBJ_LIST_FN query function fills two arrays with information about the objects currently assigned to the processor that share a processor subdomain border (in the communication graph of the application) with a given processor. Both arrays are allocated (and subsequently freed) by Zoltan; their size is determined by a call to a ZOLTAN_NUM_BORDER_OBJ_FN query function to get the array size. For certain Zoltan algorithms, either a ZOLTAN_BORDER_OBJ_LIST_FN query function or a ZOLTAN_FIRST_BORDER_OBJ_FN/ZOLTAN_NEXT_BORDER_OBJ_FN query-function pair must be registered; however, both query options need not be provided.

Function Type:	**ZOLTAN_BORDER_OBJ_LIST_FN_TYPE**
Arguments:	
data	Pointer to user-defined data.
num_gid_entries	The number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter NUM_GID_ENTRIES.
num_lid_entries	The number of array entries used to describe a single local ID. This value is the maximum value over all processors of the parameter NUM_LID_ENTRIES.
nbor_proc	The processor ID of the processor for which border objects should be returned.

global_ids	Upon return, an array of unique global IDs for all objects assigned to the processor that share a subdomain border with *nbor_proc*.
local_ids	Upon return, an array of local IDs, the meaning of which can be determined by the application, for all objects assigned to the processor that share a subdomain border with *nbor_proc*.
wgt_dim	The number of weights associated with an object (typically 1), or 0 if weights are not requested. This value is set through the parameter OBJ_WEIGHT_DIM.
obj_wgts	Upon return, an array of object weights. Weights for object *i* are stored in *obj_wgts[(i-1)*wgt_dim:i*wgt_dim-1]*. If *wgt_dim=0, obj_wgts* is undefined and may be NULL.
ierr	Error code to be set by function.

C: typedef int **ZOLTAN_FIRST_BORDER_OBJ_FN** (void **data*, int *num_gid_entries*, int *num_lid_entries*, int *nbor_proc*, **ZOLTAN_ID_PTR** *first_global_id*, **ZOLTAN_ID_PTR** *first_local_id*, int *wgt_dim*, float **first_obj_wgt*, int **ierr*);

FORTRAN: FUNCTION ***Get_First_Border_Obj***(*data, num_gid_entries, num_lid_entries, nbor_proc, first_global_id, first_local_id, wgt_dim, first_obj_wgt, ierr*)

```
INTEGER(Zoltan_INT) :: Get_First_Border_Obj
<type-data>, INTENT(IN) :: data
INTEGER(Zoltan_INT), INTENT(IN) :: num_gid_entries, num_lid_entries
INTEGER(Zoltan_INT), INTENT(IN) :: nbor_proc
INTEGER(Zoltan_INT), INTENT(OUT), DIMENSION(*) :: first_global_id
INTEGER(Zoltan_INT), INTENT(OUT), DIMENSION(*) :: first_local_id
INTEGER(Zoltan_INT), INTENT(IN) :: wgt_dim
REAL(Zoltan_FLOAT), INTENT(OUT), DIMENSION(*) :: first_obj_wgt
INTEGER(Zoltan_INT), INTENT(OUT) :: ierr
```

<*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.

A ZOLTAN_FIRST_BORDER_OBJ_FN query function initializes an iteration over objects assigned to the processor that share a processor subdomain border with a given processor. It returns the global and local IDs of the first object on the processor along the specified subdomain border. Subsequent calls to a ZOLTAN_NEXT_BORDER_OBJ_FN query function iterate over and return other objects along the requested subdomain border. This query-function pair frees the application from having to build an array of objects (as in ZOLTAN_BORDER_OBJ_LIST_FN) and allows Zoltan to obtain only as much information about objects as it needs. For some algorithms, either a ZOLTAN_BORDER_OBJ_LIST_FN query function or a ZOLTAN_FIRST_BORDER_OBJ_FN/ZOLTAN_NEXT_BORDER_OBJ_FN query-function pair must be registered; however, both query options need not be provided.

Function Type:	**ZOLTAN_FIRST_BORDER_OBJ_FN_TYPE**
Arguments:	
data	Pointer to user-defined data.
num_gid_entries	The number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter NUM_GID_ENTRIES.
num_lid_entries	The number of array entries used to describe a single local ID. This value is the maximum value over all processors of the parameter NUM_LID_ENTRIES.
nbor_proc	The processor ID of the processor for which border objects should be returned.
first_global_id	The returned value of the global ID for the first object; the value is ignored if there are no objects along the border.
first_local_id	The returned value of the local ID for the first object; the value is ignored if there are no objects along the border.
wgt_dim	The number of weights associated with an object (typically 1), or 0 if weights are not requested. This value is set through the parameter OBJ_WEIGHT_DIM.
first_obj_wgt	Upon return, the first object's weights; an array of size *wgt_dim*. Undefined if *wgt_dim*=0.
ierr	Error code to be set by function.
Returned Value:	
1	If *first_global_id* and *first_local_id* contain valid IDs of the first object along the processor border.
0	If no objects are available along this processor border.

C: typedef int **ZOLTAN_NEXT_BORDER_OBJ_FN** (void **data*, int *num_gid_entries*, int *num_lid_entries*, **ZOLTAN_ID_PTR** *global_id*, **ZOLTAN_ID_PTR** *local_id*, int *nbor_proc*, **ZOLTAN_ID_PTR** *next_global_id*, **ZOLTAN_ID_PTR** *next_local_id*, int *wgt_dim*, float **next_obj_wgt*, int **ierr*);

FORTRAN: FUNCTION ***Get_Next_Border_Obj***(*data*, *num_gid_entries*, *num_lid_entries*, *global_id*, *local_id*, *nbor_proc*, *next_global_id*, *next_local_id*, *wgt_dim*, *next_obj_wgt*, *ierr*)
INTEGER(Zoltan_INT) :: Get_Next_Border_Obj
<*type-data*>, INTENT(IN) :: data
INTEGER(Zoltan_INT), INTENT(IN) :: num_gid_entries, num_lid_entries
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: global_id
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: local_id
INTEGER(Zoltan_INT), INTENT(IN) :: nbor_proc
INTEGER(Zoltan_INT), INTENT(OUT), DIMENSION(*) :: next_global_id
INTEGER(Zoltan_INT), INTENT(OUT), DIMENSION(*) :: next_local_id
INTEGER(Zoltan_INT), INTENT(IN) :: wgt_dim
REAL(Zoltan_FLOAT), INTENT(OUT), DIMENSION(*) :: next_obj_wgt
INTEGER(Zoltan_INT), INTENT(OUT) :: ierr

<*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.

A ZOLTAN_NEXT_BORDER_OBJ_FN query function is an iterator function which, when given an object assigned to the processor and a neighboring processor ID, returns the next object assigned to the processor that shares a subdomain border with the neighboring processor. The first object of the iteration is provided by a ZOLTAN_FIRST_BORDER_OBJ_FN query function. This query-function pair frees the application from having to build an array of objects (as in ZOLTAN_BORDER_OBJ_LIST_FN) and allows Zoltan

to obtain only as much information about objects as it needs. For some algorithms, either a ZOLTAN_BORDER_OBJ_LIST_FN query function or a ZOLTAN_FIRST_BORDER_OBJ_FN/ZOLTAN_NEXT_BORDER_OBJ_FN query-function pair must be registered; however, both query options need not be provided.

Function Type:	**ZOLTAN_NEXT_BORDER_OBJ_FN_TYPE**
Arguments:	
data	Pointer to user-defined data.
num_gid_entries	The number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter NUM_GID_ENTRIES.
num_lid_entries	The number of array entries used to describe a single local ID. This value is the maximum value over all processors of the parameter NUM_LID_ENTRIES.
global_id	The global ID of the previous object.
local_id	The local ID of the previous object.
nbor_proc	The processor ID of the processor for which border objects should be returned.
next_global_id	The returned value of the global ID for the next object; the value is ignored if there are no more objects along the border.
next_local_id	The returned value of the local ID for the next object; the value is ignored if there are no more objects along the border.
wgt_dim	The number of weights associated with an object (typically 1), or 0 if weights are not requested. This value is set through the parameter OBJ_WEIGHT_DIM.
next_obj_wgt	Upon return, the weights for the next object; an array of size *wgt_dim*. Undefined if *wgt_dim*=0.
ierr	Error code to be set by function.
Returned Value:	
1	If *next_global_id* and *next_local_id* contain valid IDs of the next object along the processor border.
0	If no more objects are available along this processor border.

[Table of Contents | Next: Migration Query Functions | Previous: Application-Registered Query Functions]

Migration Query Functions

The following query functions must be registered to use any of the migration tools described in Migration Functions:

> **ZOLTAN_OBJ_SIZE_FN** or **ZOLTAN_OBJ_SIZE_MULTI_FN**
> **ZOLTAN_PACK_OBJ_FN** or **ZOLTAN_PACK_OBJ_MULTI_FN**
> **ZOLTAN_UNPACK_OBJ_FN** or **ZOLTAN_UNPACK_OBJ_MULTI_FN**

The "MULTI_" versions of the packing/unpacking functions take lists of IDs as input and pack/unpack data for all objects in the lists. Only one function of each type must be provided (e.g., either a **ZOLTAN_PACK_OBJ_FN** or **ZOLTAN_PACK_OBJ_MULTI_FN**, but not both).

Optional, additional query functions for migration may also be registered; these functions are called at the beginning, middle, and end of migration in **Zoltan_Migrate**.

> **ZOLTAN_PRE_MIGRATE_PP_FN**
> **ZOLTAN_MID_MIGRATE_PP_FN**
> **ZOLTAN_POST_MIGRATE_PP_FN**

For backward compatibility with previous versions of Zoltan, the following functions may be used with **Zoltan_Help_Migrate**.

> **ZOLTAN_PRE_MIGRATE_FN**
> **ZOLTAN_MID_MIGRATE_FN**
> **ZOLTAN_POST_MIGRATE_FN**

C and C++: typedef int **ZOLTAN_OBJ_SIZE_FN**(
void **data*,
int *num_gid_entries*,
int *num_lid_entries*,
ZOLTAN_ID_PTR *global_id*,
ZOLTAN_ID_PTR *local_id*,
int **ierr*);

FORTRAN: FUNCTION ***Obj_Size***(*data*, *num_gid_entries*, *num_lid_entries*, *global_id*, *local_id*, *ierr*)
INTEGER(Zoltan_INT) :: Obj_Size
<*type-data*>, INTENT(IN) :: data
INTEGER(Zoltan_INT), INTENT(IN) :: num_gid_entries, num_lid_entries
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: global_id, local_id
INTEGER(Zoltan_INT), INTENT(OUT) :: ierr

<*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.

A **ZOLTAN_OBJ_SIZE_FN** query function returns the size (in bytes) of the data buffer that is needed to pack all of a single object's data.

Function Type:	**ZOLTAN_OBJ_SIZE_FN_TYPE**
Arguments:	
data	Pointer to user-defined data.
num_gid_entries	The number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter NUM_GID_ENTRIES.
num_lid_entries	The number of array entries used to describe a single local ID. This value is the maximum value over all processors of the parameter NUM_LID_ENTRIES.
global_id	Pointer to the global ID of the object.
local_id	Pointer to the local ID of the object.
ierr	Error code to be set by function.
Returned Value:	
int	The size (in bytes) of the required data buffer.

C and C++: typedef void **ZOLTAN_OBJ_SIZE_MULTI_FN** (
void **data*,
int *num_gid_entries*,
int *num_lid_entries*,
int *num_ids*,
ZOLTAN_ID_PTR *global_ids*,
ZOLTAN_ID_PTR *local_ids*,
int **sizes*,
int **ierr*);

FORTRAN:	SUBROUTINE ***Obj_Size_Multi***(*data, num_gid_entries, num_lid_entries, num_ids, global_ids, local_ids, sizes, ierr*) <*type-data*>, INTENT(IN) :: data INTEGER(Zoltan_INT), INTENT(IN) :: num_gid_entries, num_lid_entries, num_ids INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: global_ids, local_ids INTEGER(Zoltan_INT), INTENT(OUT), DIMENSION(*) :: sizes INTEGER(Zoltan_INT), INTENT(OUT) :: ierr <*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.

A **ZOLTAN_OBJ_SIZE_MULTI_FN** query function is the multiple-ID version of **ZOLTAN_OBJ_SIZE_FN**. For a list of objects, it returns the per-objects sizes (in bytes) of the data buffers needed to pack object data.

Function Type:	**ZOLTAN_OBJ_SIZE_MULTI_FN_TYPE**
Arguments:	
data	Pointer to user-defined data.
num_gid_entries	The number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter NUM_GID_ENTRIES.
num_lid_entries	The number of array entries used to describe a single local ID. This value is the maximum value over all processors of the parameter NUM_LID_ENTRIES.
num_ids	The number of objects whose sizes are to be returned.
global_ids	An array of global IDs of the objects. The ID for the *i*-th object begins in *global_ids[i*num_gid_entries]*.
local_ids	An array of local IDs of the objects. The ID for the *i*-th object begins in *local_ids[i*num_lid_entries]*.
sizes	Upon return, array of sizes (in bytes) for each object in the ID lists.
ierr	Error code to be set by function.
Returned Value:	
int	The size (in bytes) of the required data buffer.

C and C++:	typedef void **ZOLTAN_PACK_OBJ_FN** (void **data*, int *num_gid_entries*, int *num_lid_entries*, **ZOLTAN_ID_PTR** *global_id*, **ZOLTAN_ID_PTR** *local_id*, int *dest*, int *size*, char **buf*, int **ierr*);
FORTRAN:	SUBROUTINE ***Pack_Obj***(*data, num_gid_entries, num_lid_entries, global_id, local_id, dest, size, buf, ierr*) <*type-data*>, INTENT(IN) :: data INTEGER(Zoltan_INT), INTENT(IN) :: num_gid_entries, num_lid_entries INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: global_id INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: local_id INTEGER(Zoltan_INT), INTENT(IN) :: dest, size INTEGER(Zoltan_INT), INTENT(OUT), DIMENSION(*) :: buf INTEGER(Zoltan_INT), INTENT(OUT) :: ierr <*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.

A **ZOLTAN_PACK_OBJ_FN** query function allows the application to tell Zoltan how to copy all needed data for a given object into a communication buffer. The object's data can then be sent to another processor as part of data migration. It may also perform other operations, such as removing the object from the processor's data structure. This routine is called by **Zoltan_Migrate** for each object to be sent to another processor.

Function Type:	**ZOLTAN_PACK_OBJ_FN_TYPE**
Arguments:	
data	Pointer to user-defined data.
num_gid_entries	The number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter NUM_GID_ENTRIES.
num_lid_entries	The number of array entries used to describe a single local ID. This value is the maximum value over all processors of the parameter NUM_LID_ENTRIES.
global_id	The global ID of the object for which data should be copied into the communication buffer.
local_id	The local ID of the object for which data should be copied into the communication buffer.
dest	The destination partition (i.e., the partition to which the object is being sent)

size	The size (in bytes) of the communication buffer for the specified object (as returned by the **ZOLTAN_OBJ_SIZE_FN** query function).
buf	The starting address of the communication buffer into which the object's data should be packed.
ierr	Error code to be set by function.

C and C++: typedef void **ZOLTAN_PACK_OBJ_MULTI_FN** (
void **data*,
int *num_gid_entries*,
int *num_lid_entries*,
int *num_ids*,
ZOLTAN_ID_PTR *global_ids*,
ZOLTAN_ID_PTR *local_ids*,
int **dest*,
int **sizes*,
int **idx*,
char **buf*,
int **ierr*);

FORTRAN: SUBROUTINE ***Pack_Obj_Multi***(*data, num_gid_entries, num_lid_entries, num_ids, global_ids, local_ids, dest, sizes, idx, buf, ierr*)
<*type-data*>, INTENT(IN) :: data
INTEGER(Zoltan_INT), INTENT(IN) :: num_gid_entries, num_lid_entries, num_ids
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: global_ids
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: local_ids
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: dest
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: sizes
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: idx
INTEGER(Zoltan_INT), INTENT(OUT), DIMENSION(*) :: buf
INTEGER(Zoltan_INT), INTENT(OUT) :: ierr

<*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.

A **ZOLTAN_PACK_OBJ_MULTI_FN** query function is the multiple-ID version of a **ZOLTAN_PACK_OBJ_FN**. It allows the application to tell Zoltan how to copy all needed data for a given list of objects into a communication buffer.

Function Type:	**ZOLTAN_PACK_OBJ_FN_MULTI_TYPE**
Arguments:	
data	Pointer to user-defined data.

num_gid_entries	The number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter NUM_GID_ENTRIES.
num_lid_entries	The number of array entries used to describe a single local ID. This value is the maximum value over all processors of the parameter NUM_LID_ENTRIES.
num_ids	The number of objects to be packed.
global_ids	An array of global IDs of the objects. The ID for the *i*-th object begins in *global_ids[i*num_gid_entries]*.
local_ids	An array of local IDs of the objects. The ID for the *i*-th object begins in *local_ids[i*num_lid_entries]*.
dest	An array of destination partition numbers (i.e., the partitions to which the objects are being sent)
sizes	An array containing the per-object sizes (in bytes) of the communication buffer for each object.
idx	For each object, an index into the *buf* array giving the starting location of that object's data. Data for the *i*-th object are stored in *buf[idx[i]]*, *buf[idx[i]+1]*, ..., *buf[idx[i]+sizes[i]-1]*. Because Zoltan adds some tag information to packed data, *idx[i]* != sum[*j*=0,*i*-1](*sizes[j]*).
buf	The address of the communication buffer into which the objects' data should be packed.
ierr	Error code to be set by function.

C and C++: typedef void **ZOLTAN_UNPACK_OBJ_FN** (
 void **data*,
 int *num_gid_entries*,
 ZOLTAN_ID_PTR *global_id*,
 int *size*,
 char **buf*,
 int **ierr*);

FORTRAN: SUBROUTINE ***Unpack_Obj***(*data*, *num_gid_entries*, *global_id*, *size*, *buf*, *ierr*)
<*type-data*>, INTENT(INOUT) :: data
INTEGER(Zoltan_INT), INTENT(IN) :: num_gid_entries
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: global_id
INTEGER(Zoltan_INT), INTENT(IN) :: size
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: buf
INTEGER(Zoltan_INT), INTENT(OUT) :: ierr

<*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.

A **ZOLTAN_UNPACK_OBJ_FN** query function allows the application to tell Zoltan how to copy all needed data for a given object from a communication buffer into the application's data structure. This operation is needed as the final step of importing objects during data migration. The query function may also perform other computation, such as building request lists for related data. This routine is called by **Zoltan_Migrate** for each object to be received by the processor. (Note: a local ID for the object is not included in this function, as the local ID is local to the exporting, not the importing, processor.)

Function Type:	**ZOLTAN_UNPACK_OBJ_FN_TYPE**
Arguments:	
data	Pointer to user-defined data.
num_gid_entries	The number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter NUM_GID_ENTRIES.
global_id	The global ID of the object whose data has been received in the communication buffer.
size	The size (in bytes) of the object's data in the communication buffer.
buf	The starting address of the communication buffer for this object.
ierr	Error code to be set by function.

C and C++: typedef void **ZOLTAN_UNPACK_OBJ_MULTI_FN** (
void **data*,
int *num_gid_entries*,
int *num_ids*,
ZOLTAN_ID_PTR *global_ids*,
int **sizes*,
int **idx*,
char **buf*,
int **ierr*);

FORTRAN: SUBROUTINE ***Unpack_Obj_Multi***(*data, num_gid_entries, num_ids, global_ids, sizes, idx, buf, ierr*)
<*type-data*>, INTENT(INOUT) :: data
INTEGER(Zoltan_INT), INTENT(IN) :: num_gid_entries
INTEGER(Zoltan_INT), INTENT(IN) :: num_ids
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: global_ids
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: sizes
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: idx
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: buf
INTEGER(Zoltan_INT), INTENT(OUT) :: ierr

<*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.

A **ZOLTAN_UNPACK_OBJ_MULTI_FN** query function is the multiple-ID version of a **ZOLTAN_UNPACK_OBJ_FN**. It allows the application to tell Zoltan how to copy all needed data for a given list of objects from a communication buffer into the application's data structure.

Function Type:	**ZOLTAN_UNPACK_OBJ_MULTI_FN_TYPE**
Arguments:	
data	Pointer to user-defined data.
num_gid_entries	The number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter NUM_GID_ENTRIES.
num_ids	The number of objects to be unpacked.
global_ids	An array of global IDs of the objects. The ID for the *i*-th object begins in *global_ids[i*num_gid_entries]*.
sizes	An array containing the per-object sizes (in bytes) of the communication buffer for each object.
idx	For each object, an index into the *buf* array giving the starting location of that object's data. Data for the *i*-th object are stored in *buf[idx[i]]*, *buf[idx[i]+1]*, ..., *buf[idx[i]+sizes[i]-1]*. Because Zoltan adds some tag information to packed data, *idx[i]* != sum[*j*=0,*i*-1](*sizes[j]*).
buf	The address of the communication buffer from which data is unpacked.
ierr	Error code to be set by function.

C and C++: typedef void **ZOLTAN_PRE_MIGRATE_PP_FN** (
void **data*,
int *num_gid_entries*,
int *num_lid_entries*,
int *num_import*,
ZOLTAN_ID_PTR *import_global_ids*,
ZOLTAN_ID_PTR *import_local_ids*,
int **import_procs*,
int **import_to_part*,
int *num_export*,
ZOLTAN_ID_PTR *export_global_ids*,
ZOLTAN_ID_PTR *export_local_ids*,
int **export_procs*,
int **export_to_part*,
int **ierr*);

FORTRAN: SUBROUTINE ***Pre_Migrate_PP***(*data, num_gid_entries, num_lid_entries, num_import, import_global_ids, import_local_ids, import_procs, import_to_part, num_export, export_global_ids, export_local_ids, export_procs, export_to_part, ierr*)
<*type-data*>, INTENT(INOUT) :: data
INTEGER(Zoltan_INT), INTENT(IN) :: num_gid_entries, num_lid_entries
INTEGER(Zoltan_INT), INTENT(IN) :: num_import, num_export
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: import_global_ids, export_global_ids
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: import_local_ids, export_local_ids
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: import_procs, export_procs
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: import_to_part, export_to_part
INTEGER(Zoltan_INT), INTENT(OUT) :: ierr

<*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.

A **ZOLTAN_PRE_MIGRATE_PP_FN** query function performs any pre-processing desired by the application. If it is registered, it is called at the beginning of the **Zoltan_Migrate** routine. The arguments passed to **Zoltan_Migrate** are made available for use in the pre-processing routine.

Function Type: **ZOLTAN_PRE_MIGRATE_PP_FN_TYPE**

Arguments:

data	Pointer to user-defined data.
num_gid_entries	The number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter NUM_GID_ENTRIES.
num_lid_entries	The number of array entries used to describe a single local ID. This value is the maximum value over all processors of the parameter NUM_LID_ENTRIES.
num_import	The number of objects that will be received by this processor.
import_global_ids	An array of *num_import* global IDs of objects to be received by this processor. This array may be NULL, as the processor does not necessarily need to know which objects it will receive.
import_local_ids	An array of *num_import* local IDs of objects to be received by this processor. This array may be NULL, as the processor does not necessarily need to know which objects it will receive.
import_procs	An array of size *num_import* listing the processor IDs of the source processors. This array may be NULL, as the processor does not necessarily need to know which objects is will receive.
import_to_part	An array of size *num_import* listing the partitions to which objects will be imported. This array may be NULL, as the processor does not necessarily need to know from which objects it will receive.

num_export	The number of objects that will be sent from this processor to other processors.
export_global_ids	An array of *num_export* global IDs of objects to be sent from this processor.
export_local_ids	An array of *num_export* local IDs of objects to be sent from this processor.
export_procs	An array of size *num_export* listing the processor IDs of the destination processors.
export_to_part	An array of size *num_export* listing the partitions to which objects will be sent.
ierr	Error code to be set by function.

Default:

No pre-processing is done if a **ZOLTAN_PRE_MIGRATE_PP_FN** is not registered.

C and C++: typedef void **ZOLTAN_MID_MIGRATE_PP_FN** (
void **data*,
int *num_gid_entries*,
int *num_lid_entries*,
int *num_import*,
ZOLTAN_ID_PTR *import_global_ids*,
ZOLTAN_ID_PTR *import_local_ids*,
int **import_procs*,
int **import_to_part*,
int *num_export*,
ZOLTAN_ID_PTR *export_global_ids*,
ZOLTAN_ID_PTR *export_local_ids*,
int **export_procs*,
int **export_to_part*,
int **ierr*);

FORTRAN: SUBROUTINE ***Mid_Migrate_PP***(*data*, *num_gid_entries*, *num_lid_entries*, *num_import*, *import_global_ids*, *import_local_ids*, *import_procs*, *import_to_part*, *num_export*, *export_global_ids*, *export_local_ids*, *export_procs*, *export_to_part*, *ierr*)
<*type-data*>, INTENT(INOUT) :: data
INTEGER(Zoltan_INT), INTENT(IN) :: num_gid_entries, num_lid_entries
INTEGER(Zoltan_INT), INTENT(IN) :: num_import, num_export
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: import_global_ids, export_global_ids
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: import_local_ids, export_local_ids
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: import_procs, export_procs
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: import_to_part, export_to_part
INTEGER(Zoltan_INT), INTENT(OUT) :: ierr

<*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section

on Fortran query functions for an explanation.

A **ZOLTAN_MID_MIGRATE_PP_FN** query function performs any processing desired by the application between the packing and unpacking of objects being migrated. If it is registered, it is called after export objects are packed in **Zoltan_Migrate**; imported objects are unpacked after the **ZOLTAN_MID_MIGRATE_PP_FN** query function is called. The arguments passed to **Zoltan_Migrate** are made available for use in the processing routine.

,

Function Type:	**ZOLTAN_MID_MIGRATE_PP_FN_TYPE**
Arguments:	
data	Pointer to user-defined data.
num_gid_entries	The number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter NUM_GID_ENTRIES.
num_lid_entries	The number of array entries used to describe a single local ID. This value is the maximum value over all processors of the parameter NUM_LID_ENTRIES.
num_import	The number of objects that will be received by this processor.
import_global_ids	An array of *num_import* global IDs of objects to be received by this processor. This array may be NULL, as the processor does not necessarily need to know which objects it will receive.
import_local_ids	An array of *num_import* local IDs of objects to be received by this processor. This array may be NULL, as the processor does not necessarily need to know which objects it will receive.
import_procs	An array of size *num_import* listing the processor IDs of the source processors. This array may be NULL, as the processor does not necessarily need to know which objects is will receive.
import_to_part	An array of size *num_import* listing the partitions to which objects will be imported. This array may be NULL, as the processor does not necessarily need to know from which objects it will receive.
num_export	The number of objects that will be sent from this processor to other processors.
export_global_ids	An array of *num_export* global IDs of objects to be sent from this processor.
export_local_ids	An array of *num_export* local IDs of objects to be sent from this processor.
export_procs	An array of size *num_export* listing the processor IDs of the destination processors.
export_to_part	An array of size *num_export* listing the partitions to which objects will be sent.
ierr	Error code to be set by function.
Default:	
	No processing is done if a **ZOLTAN_MID_MIGRATE_PP_FN** is not registered.

C and C++: typedef void **ZOLTAN_POST_MIGRATE_PP_FN** (
void **data*,
int *num_gid_entries*,
int *num_lid_entries*,
int *num_import*,
ZOLTAN_ID_PTR *import_global_ids*,
ZOLTAN_ID_PTR *import_local_ids*,
int **import_procs*,
int **import_to_part*,
int *num_export*,
ZOLTAN_ID_PTR *export_global_ids*,
ZOLTAN_ID_PTR *export_local_ids*,
int **export_procs*,
int **export_to_part*,
int **ierr*);

FORTRAN: SUBROUTINE ***Post_Migrate_PP***(*data*, *num_gid_entries*, *num_lid_entries*, *num_import*, *import_global_ids*, *import_local_ids*, *import_procs*, *import_to_part*, *num_export*, *export_global_ids*, *export_local_ids*, *export_procs*, *export_to_part*, *ierr*)
<*type-data*>, INTENT(INOUT) :: data
INTEGER(Zoltan_INT), INTENT(IN) :: num_gid_entries, num_lid_entries
INTEGER(Zoltan_INT), INTENT(IN) :: num_import, num_export
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: import_global_ids, export_global_ids
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: import_local_ids, export_local_ids
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: import_procs, export_procs
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: import_to_part, export_to_part
INTEGER(Zoltan_INT), INTENT(OUT) :: ierr

<*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.

A **ZOLTAN_POST_MIGRATE_PP_FN** query function performs any post-processing desired by the application. If it is registered, it is called at the end of the **Zoltan_Migrate** routine. The arguments passed to **Zoltan_Migrate** are made available for use in the post-processing routine.

Function Type:	**ZOLTAN_POST_MIGRATE_PP_FN_TYPE**
Arguments:	
data	Pointer to user-defined data.
num_gid_entries	The number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter NUM_GID_ENTRIES.

num_lid_entries	The number of array entries used to describe a single local ID. This value is the maximum value over all processors of the parameter NUM_LID_ENTRIES.
num_import	The number of objects that will be received by this processor.
import_global_ids	An array of *num_import* global IDs of objects to be received by this processor. This array may be NULL, as the processor does not necessarily need to know which objects it will receive.
import_local_ids	An array of *num_import* local IDs of objects to be received by this processor. This array may be NULL, as the processor does not necessarily need to know which objects it will receive.
import_procs	An array of size *num_import* listing the processor IDs of the source processors. This array may be NULL, as the processor does not necessarily need to know which objects is will receive.
import_to_part	An array of size *num_import* listing the partitions to which objects will be imported. This array may be NULL, as the processor does not necessarily need to know from which objects it will receive.
num_export	The number of objects that will be sent from this processor to other processors.
export_global_ids	An array of *num_export* global IDs of objects to be sent from this processor.
export_local_ids	An array of *num_export* local IDs of objects to be sent from this processor.
export_procs	An array of size *num_export* listing the processor IDs of the destination processors.
export_to_part	An array of size *num_export* listing the partitions to which objects will be sent.
ierr	Error code to be set by function.

Default:

No post-processing is done if a **ZOLTAN_POST_MIGRATE_PP_FN** is not registered.

C: typedef void **ZOLTAN_PRE_MIGRATE_FN** (
void **data*,
int *num_gid_entries*,
int *num_lid_entries*,
int *num_import*,
ZOLTAN_ID_PTR *import_global_ids*,
ZOLTAN_ID_PTR *import_local_ids*,
int **import_procs*,
int *num_export*,
ZOLTAN_ID_PTR *export_global_ids*,
ZOLTAN_ID_PTR *export_local_ids*,
int **export_procs*,
int **ierr*);

FORTRAN: SUBROUTINE ***Pre_Migrate***(*data*, *num_gid_entries*, *num_lid_entries*, *num_import*, *import_global_ids*, *import_local_ids*, *import_procs*, *num_export*, *export_global_ids*, *export_local_ids*, *export_procs*, *ierr*)
<*type-data*>, INTENT(INOUT) :: data
INTEGER(Zoltan_INT), INTENT(IN) :: num_gid_entries, num_lid_entries
INTEGER(Zoltan_INT), INTENT(IN) :: num_import, num_export
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: import_global_ids, export_global_ids
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: import_local_ids, export_local_ids
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: import_procs, export_procs
INTEGER(Zoltan_INT), INTENT(OUT) :: ierr

<*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.

A **ZOLTAN_PRE_MIGRATE_FN** query function performs any pre-processing desired by applications using **Zoltan_Help_Migrate**. Its function is analogous to **ZOLTAN_PRE_MIGRATE_PP_FN**, but it cannot be used with **Zoltan_Migrate**.

Function Type: **ZOLTAN_PRE_MIGRATE_FN_TYPE**

Arguments:

All arguments are analogous to those in **ZOLTAN_PRE_MIGRATE_PP_FN**. Partition-assignment arguments *import_to_part* and *export_to_part* are not included, as processor and partitions numbers are considered to be the same in **Zoltan_Help_Migrate**.

Default:

No pre-processing is done if a **ZOLTAN_PRE_MIGRATE_FN** is not registered.

C: typedef void **ZOLTAN_MID_MIGRATE_FN** (
void **data*,
int *num_gid_entries*,
int *num_lid_entries*,
int *num_import*,
ZOLTAN_ID_PTR *import_global_ids*,
ZOLTAN_ID_PTR *import_local_ids*,
int **import_procs*,
int *num_export*,
ZOLTAN_ID_PTR *export_global_ids*,
ZOLTAN_ID_PTR *export_local_ids*,
int **export_procs*,

int **ierr*);

FORTRAN: SUBROUTINE ***Mid_Migrate***(*data*, *num_gid_entries*, *num_lid_entries*, *num_import*, *import_global_ids*, *import_local_ids*, *import_procs*, *num_export*, *export_global_ids*, *export_local_ids*, *export_procs*, *ierr*)
<*type-data*>, INTENT(INOUT) :: data
INTEGER(Zoltan_INT), INTENT(IN) :: num_gid_entries, num_lid_entries
INTEGER(Zoltan_INT), INTENT(IN) :: num_import, num_export
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: import_global_ids, export_global_ids
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: import_local_ids, export_local_ids
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: import_procs, export_procs
INTEGER(Zoltan_INT), INTENT(OUT) :: ierr

<*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.

A **ZOLTAN_MID_MIGRATE_FN** query function performs any mid-migration processing desired by applications using **Zoltan_Help_Migrate**. Its function is analogous to **ZOLTAN_MID_MIGRATE_PP_FN**, but it cannot be used with **Zoltan_Migrate**.

Function Type: **ZOLTAN_MID_MIGRATE_FN_TYPE**

Arguments:

All arguments are analogous to those in **ZOLTAN_MID_MIGRATE_PP_FN**. Partition-assignment arguments *import_to_part* and *export_to_part* are not included, as processor and partitions numbers are considered to be the same in **Zoltan_Help_Migrate**.

Default:

No processing is done if a **ZOLTAN_MID_MIGRATE_FN** is not registered.

C: typedef void **ZOLTAN_POST_MIGRATE_FN** (
void **data*,
int *num_gid_entries*,
int *num_lid_entries*,
int *num_import*,
ZOLTAN_ID_PTR *import_global_ids*,
ZOLTAN_ID_PTR *import_local_ids*,
int **import_procs*,
int *num_export*,
ZOLTAN_ID_PTR *export_global_ids*,
ZOLTAN_ID_PTR *export_local_ids*,
int **export_procs*,
int **ierr*);

FORTRAN: SUBROUTINE ***Post_Migrate***(*data, num_gid_entries, num_lid_entries, num_import, import_global_ids, import_local_ids, import_procs, num_export, export_global_ids, export_local_ids, export_procs, ierr*)
<*type-data*>, INTENT(INOUT) :: data
INTEGER(Zoltan_INT), INTENT(IN) :: num_gid_entries, num_lid_entries
INTEGER(Zoltan_INT), INTENT(IN) :: num_import, num_export
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: import_global_ids, export_global_ids
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: import_local_ids, export_local_ids
INTEGER(Zoltan_INT), INTENT(IN), DIMENSION(*) :: import_procs, export_procs
INTEGER(Zoltan_INT), INTENT(OUT) :: ierr

<*type-data*> can be any of INTEGER(Zoltan_INT), DIMENSION(*) or REAL(Zoltan_FLOAT), DIMENSION(*) or REAL(Zoltan_DOUBLE), DIMENSION(*) or TYPE(Zoltan_User_Data_*x*) where *x* is 1, 2, 3 or 4. See the section on Fortran query functions for an explanation.

A **ZOLTAN_POST_MIGRATE_FN** query function performs any post-processing desired by applications using **Zoltan_Help_Migrate**. Its function is analogous to **ZOLTAN_POST_MIGRATE_PP_FN**, but it cannot be used with **Zoltan_Migrate**.

Function Type: **ZOLTAN_POST_MIGRATE_FN_TYPE**

Arguments:

All arguments are analogous to those in **ZOLTAN_POST_MIGRATE_PP_FN**. Partition-assignment arguments *import_to_part* and *export_to_part* are not included, as processor and partitions numbers are considered to be the same in **Zoltan_Help_Migrate**.

Default:

No post-processing is done if a **ZOLTAN_POST_MIGRATE_FN** is not registered.

[Table of Contents | Next: Zoltan Parameters and Output Levels | Previous: Load-Balancing Query Functions]

Zoltan Parameters and Output Levels

The behavior of Zoltan is controlled by several parameters and debugging-output levels. These parameters can be set by calls to **Zoltan_Set_Param**. Reasonable default values for all parameters are specified by Zoltan. Many of the parameters are specific to individual algorithms, and are listed in the descriptions of those algorithms. However, the parameters below have meaning across the entire library.

General Parameters

The following parameters apply to the entire Zoltan library. While reasonable default values for all parameters are specified by Zoltan, applications can change these values through calls to **Zoltan_Set_Param**.

Parameters:

NUM_GID_ENTRIES	The number of unsigned integers that should be used to represent a global identifier (ID). Values greater than zero are accepted.
NUM_LID_ENTRIES	The number of unsigned integers that should be used to represent a local identifier (ID). Values greater than or equal to zero are accepted.
DEBUG_LEVEL	An integer indicating how much debugging information is printed by Zoltan. Higher values of *DEBUG_LEVEL* produce more output and potentially slow down Zoltan's computations. The least output is produced when *DEBUG_LEVEL*= 0. *DEBUG_LEVEL* primarily controls Zoltan's behavior; most algorithms have their own parameters to control their output level. Values used within Zoltan are listed below. Note: Because some debugging levels use processor synchronization, all processors should use the same value of *DEBUG_LEVEL*.
DEBUG_PROCESSOR	Processor number from which trace output should be printed when *DEBUG_LEVEL* is 5.
DEBUG_MEMORY	Integer indicating the amount of low-level debugging information about memory-allocation should be kept by Zoltan's Memory Management utilities. Valid values are 0, 1, 2, and 3.
OBJ_WEIGHT_DIM	The number of weights associated with an object. If this parameter is zero, all objects have equal weight. Some algorithms may not support multiple (multidimensional) weights.
EDGE_WEIGHT_DIM	The number of weights associated with an edge. If this parameter is zero, all edges have equal weight. Many algorithms do not support multiple (multidimensional) weights.

TIMER	The timer with which you wish to measure time. Valid choices are *wall* (based on *MPI_Wtime), cpu* (based on the ANSI C library function *clock*), and *user*. The resolution may be poor, as low as 1/60th of a second, depending upon your platform.
USE_MACHINE_DESC	Currently unused; will be used when heterogeneous computers are supported.
MACHINE_DESC_FILE	Currently unused; will be used when heterogeneous computers are supported.

Default Values:

NUM_GID_ENTRIES = 1
NUM_LID_ENTRIES = 1
DEBUG_LEVEL = 1
DEBUG_PROCESSOR = 0
DEBUG_MEMORY = 1
OBJ_WEIGHT_DIM = 0
EDGE_WEIGHT_DIM = 0
TIMER = wall
USE_MACHINE_DESC = 0
MACHINE_DESC_FILE = /etc/local/Zoltan_Machine_Desc

Debugging Levels in Zoltan

The *DEBUG_LEVEL* parameter determines how much debugging information is printed to *stdout* by Zoltan. It is set by a call to **Zoltan_Set_Param**. Higher values of *DEBUG_LEVEL* produce more output and can slow down Zoltan's computations, especially when the output is printed by one processor at a time. The least output is produced when *DEBUG_LEVEL* = 0.

Descriptions of the output produced by Zoltan for each value of *DEBUG_LEVEL* are included below. For a given *DEBUG_LEVEL* value *n*, all output for values less than or equal to *n* is produced.

Some high debugging levels use processor synchronization to force processors to write one-at-a-time. For example, when *DEBUG_LEVEL* is greater than or equal to eight, each processor writes its list in turn so that the lists from all processors are not jumbled together in the output. This synchronization requires all processors to use the same value of *DEBUG_LEVEL*.

DEBUG_LEVEL	**Output Produced**
0	Quiet mode; no output unless an error or warning is produced.
1	Values of all parameters set by **Zoltan_Set_Param** and used by Zoltan.
2	Timing information for Zoltan's main routines.
3	Timing information within Zoltan's algorithms (support by algorithms is optional).
4	
5	Trace information (enter/exit) for major Zoltan interface routines (printed by the processor specified by the *DEBUG_PROCESSOR* parameter).

6	Trace information (enter/exit) for major Zoltan interface routines (printed by all processors).
7	More detailed trace information in major Zoltan interface routines.
8	List of objects to be imported to and exported from each processor. [1]
9	
10	Maximum debug output; may include algorithm-specific output. [1]

[1] *Output may be serialized; that is, one processor may have to complete its output before the next processor is allowed to begin its output. This serialization is not scalable and can significantly increase execution time on large number of processors.*

[Table of Contents | Next: Load-Balancing Algorithms | Previous: Migration Query Functions]

Load-Balancing Algorithms

The following dynamic load-balancing algorithms are currently included in the Zoltan library:

- Recursive Coordinate Bisection (RCB)
- Recursive Inertial Bisection (RIB)
- Hilbert Space-Filling Curve (HSFC)
- Refinement Tree Based Partitioning (REFTREE)
- Graph partitioning: ParMETIS (GRAPH or PARMETIS)
- Graph partitioning: Jostle (JOSTLE)
- Hypergraph partitioning: PHG (HYPERGRAPH or PHG)
- Octree Partitioning (OCTPART)

The parenthetical string is the parameter value for *LB_METHOD* parameter; the parameter is set through a call to **Zoltan_Set_Param**.

For further analysis and discussion of the algorithms, see [Hendrickson and Devine].

Load-Balancing Parameters

While the overall behavior of Zoltan is controlled by general Zoltan parameters, the behavior of each load-balancing method is controlled by parameters specific to partitioning which are also set by calls to **Zoltan_Set_Param**. Many of these parameters are specific to individual partitioning algorithms, and are listed in the descriptions of the individual algorithms. However, several have meaning across multiple partitioning algorithms. These load-balancing parameters are described below. Unless indicated otherwise, these parameters apply to both **Zoltan_LB_Partition** and **Zoltan_LB_Balance**.

If any processor sets this parameter, NUM_LOCAL_PARTITIONS is assumed to be zero on processors not setting this parameter.

Parameters:

LB_METHOD	The load-balancing algorithm used by Zoltan is specified by this parameter. Valid values are "RCB" (for recursive coordinate bisection), "RIB" (for recursive inertial bisection), "HSFC" (for Hilbert space-filling curve partitioning), "GRAPH" or "PARMETIS" (for any of the methods in the ParMETIS library), "JOSTLE" (for any of the methods in the Jostle library), "HYPERGRAPH" or "PHG" (for hypergraph partitioning), "OCTPART" (for octree partitioning), "REFTREE" (for refinement tree based partitioning), and "NONE" (for no load-balancing).
NUM_GLOBAL_PARTITIONS	The total number of partitions to be generated by a call to **Zoltan_LB_Partition**. Integer values greater than zero are accepted. Not valid for **Zoltan_LB_Balance**.
NUM_LOCAL_PARTITIONS	The number of partitions to be generated on this processor by a call to **Zoltan_LB_Partition**. Integer values greater than or equal to zero are accepted. Not valid for **Zoltan_LB_Balance**.
RETURN_LISTS	The lists returned by calls to **Zoltan_LB_Partition** or **Zoltan_LB_Balance**. Valid values are "IMPORT", to return only information about objects to be imported to a processor "EXPORT", to return only information about objects to be exported from a processor "ALL", or "IMPORT AND EXPORT" (or any string with both "IMPORT" and "EXPORT" in it) to return both import and export information "PARTITION ASSIGNMENTS" (or any string with "PARTITION" in it) to return the new process and partition assignment of every local object, including those not being exported. "NONE", to return neither import nor export information

REMAP	Within **Zoltan_LB_Partition** or **Zoltan_LB_Balance**, renumber partitions to maximize overlap between the old decomposition and the new decomposition (to reduce data movement from old to new decompositions). Valid values are "0" (no remapping) or "1" (remapping). Requests for remapping are ignored when, in the new decomposition, a partition is spread across multiple processors or partition sizes are specified using **Zoltan_LB_Set_Part_Sizes**.
IMBALANCE_TOL	The amount of load imbalance the partitioning algorithm should deem acceptable. The load on each processor is computed as the sum of the weights of objects it is assigned. The imbalance is then computed as the maximum load divided by the average load. An value for *IMBALANCE_TOL* of 1.2 indicates that 20% imbalance is OK; that is, the maximum over the average shouldn't exceed 1.2.
MIGRATE_ONLY_PROC_CHANGES	If this value is set to TRUE (non-zero), Zoltan's migration functions will migrate only objects moving to new processors. They will not migrate objects for which only the partition number has changed; the objects' processor numbers must change as well. If this value is set to FALSE (zero), Zoltan's migration functions will migrate all objects with new partition or processor assignments.
AUTO_MIGRATE	If this value is set to TRUE (non-zero), Zoltan will automatically perform the data migration during calls to **Zoltan_LB_Partition** or **Zoltan_LB_Balance**. A full discussion of automatic migration can be found in the description of the migration interface functions.

Default Values:

LB_METHOD = RCB
NUM_GLOBAL_PARTITIONS = Number of processors specified in **Zoltan_Create**.
NUM_LOCAL_PARTITIONS = 1
RETURN_LISTS = ALL
REMAP = 1
IMBALANCE_TOL = 1.1
MIGRATE_ONLY_PROC_CHANGES = 1
AUTO_MIGRATE = FALSE

Recursive Coordinate Bisection (RCB)

An implementation of Recursive Coordinate Bisection (RCB) due to Steve Plimpton of Sandia National Laboratories is included in Zoltan. RCB was first proposed as a static load-balancing algorithm by Berger and Bokhari, but is attractive as a dynamic load-balancing algorithm because it implicitly produces incremental partitions. In RCB, the computational domain is first divided into two regions by a cutting plane orthogonal to one of the coordinate axes so that half the work load is in each of the sub-regions. The splitting direction is determined by computing in which coordinate direction the set of objects is most elongated, based upon the geometric locations of the objects. The sub-regions are then further divided by recursive application of the same splitting algorithm until the number of sub-regions equals the number of processors. Although this algorithm was first devised to cut into a number of sets which is a power of two, the set sizes in a particular cut needn't be equal. By adjusting the partition sizes appropriately, any number of equally-sized sets can be created. If the parallel machine has processors with different speeds, sets with nonuniform sizes can also be easily generated. The Zoltan implementation of RCB has several parameters which can be modified by the **Zoltan_Set_Param** function. A recent feature is that RCB allows multiple weights; that is, one can balance with respect to several load criteria simultaneously. Note that there is no guarantee that a desired load balance tolerance can be achieved using RCB, especially in the multiconstraint case.

Information about the sub-regions generated by RCB can be obtained by an application through calls to **Zoltan_RCB_Box**. This function is not required to perform load balancing; it only provides auxiliary information to an application.

Method String:	**RCB**
Parameters:	
RCB_OVERALLOC	The amount by which to over-allocate temporary storage arrays for objects within the RCB algorithm when additional storage is due to changes in processor assignments. 1.0 = no extra storage allocated; 1.5 = 50% extra storage; etc.
RCB_REUSE	Flag to indicate whether to use previous cuts as initial guesses for the current RCB invocation. 0 = don't use previous cuts; 1 = use previous cuts.
RCB_OUTPUT_LEVEL	Flag controlling the amount of timing and diagnostic output the routine produces. 0 = no output; 1 = print summary; 2 = print data for each processor.
CHECK_GEOM	Flag controlling the invocation of input and output error checking. 0 = don't do checking; 1 = do checking.

KEEP_CUTS	Should information about the cuts determining the RCB decomposition be retained? It costs a bit of time to do so, but this information is necessary if application wants to add more objects to the decomposition via calls to **Zoltan_LB_Point_PP_Assign** or to **Zoltan_LB_Box_PP_Assign**. 0 = don't keep cuts; 1 = keep cuts.
AVERAGE_CUTS	When set to one, coordinates of RCB cutting planes are computed to be the average of the coordinates of the closest object on each side of the cut. Otherwise, coordinates of cutting planes may equal those of one of the closest objects. 0 = don't average cuts; 1 = average cuts.
RCB_LOCK_DIRECTIONS	Flag that determines whether the order of the directions of the cuts is kept constant after they are determined the first time RCB is called. 0 = don't lock directions; 1 = lock directions.
RCB_SET_DIRECTIONS	If this flag is set, the order of cuts is changed so that all of the cuts in any direction are done as a group. The number of cuts in each direction is determined and then the value of the parameter is used to determine the order that those cuts are made in. When 1D and 2D problems are partitioned, the directions corresponding to unused dimensions are ignored. 0 = don't order cuts; 1 = xyz; 2 = xzy; 3 = yzx; 4 = yxz; 5 = zxy; 6 = zyx;
RCB_RECTILINEAR_BLOCKS	Flag controlling the shape of the resulting regions. If this option is specified, then when a cut is made, all of the dots located on the cut are moved to the same side of the cut. The resulting regions are then rectilinear. When these dots are treated as a group, then the resulting load balance may not be as good as when the group of dots is split by the cut. 0 = move dots individually; 1 = move dots in groups.
REDUCE_DIMENSIONS	When a 3 dimensional geometry is almost flat, it may make more sense to treat it as a 2 dimensional geometry when applying the RCB algorithm. In this case, a 2 dimensional RCB calculation is applied to a plane that corresponds with the geometry. (This results in cuts that, while still orthogonal, may no longer be axis aligned.) If this parameter is set to **1**, a 3 dimensional geometry will be treated as 2 dimensional if it is very flat, or 1 dimensional if it is very thin. A 2 dimensional geometry will be treated as 1 dimensional if it is very thin.
DEGENERATE_RATIO	If the **REDUCE_DIMENSIONS** parameter is set, then this parameter determines when a geometry is considered to be degenerate. A bounding box which is oriented to the geometry is constructed, and the lengths of its sides are tested against a ratio of 1 : **DEGENERATE_RATIO**.

RCB_RECOMPUTE_BOX	Flag indicating whether the bounding box of set of partitions is recomputed at each level of recursion. By default, the longest direction of the bounding box is cut during bisection. Recomputing the bounding box at each level of recursion can produce more effective cut directions for unusually shaped geometries; the computation does, however, take additional time and communication, and may cause cut directions to vary from one invocation of RCB to the next. 0 = don't recompute the bounding box; 1 = recompute the box.
OBJ_WEIGHTS_COMPARABLE	In the multiconstraint case, are the object weights comparable? Do they have the same units and is the scaling meaningful? For example, if the jth weight corresponds to the expected time in phase j (measured in seconds), set this parameter to 1. (0 = incomparable, 1 = comparable)
RCB_MULTICRITERIA_NORM	Norm used in multicriteria algorithm; this determines how to balance the different weight constraints. Valid values are 1,2, and 3. Roughly, if the weights correspond to different phases, then the value 1 (1-norm) tries to minimize the total time (sum over all phases) while the value 3 (max-norm) attempts to minimize the worst imbalance in any phase. The 2-norm does something in between. Try a different value if you're not happy with the balance.
RCB_MAX_ASPECT_RATIO	Maximum allowed ratio between the largest and smallest side of a subdomain. Must be > 1.

Default:

RCB_OVERALLOC = 1.0
RCB_REUSE = 0
RCB_OUTPUT_LEVEL = 0
CHECK_GEOM = 1
KEEP_CUTS = 0
AVERAGE_CUTS = 0
RCB_LOCK_DIRECTIONS = 0
REDUCE_DIMENSIONS = 0
DEGENERATE_RATIO = 10
RCB_SET_DIRECTIONS = 0
RCB_RECTILINEAR_BLOCKS = 0
RCB_RECOMPUTE_BOX = 0
OBJ_WEIGHTS_COMPARABLE = 0
RCB_MULTICRITERIA_NORM = 1
RCB_MAX_ASPECT_RATIO = 10

Required Query Functions:

ZOLTAN_NUM_OBJ_FN
ZOLTAN_OBJ_LIST_FN or
ZOLTAN_FIRST_OBJ_FN/**ZOLTAN_NEXT_OBJ_FN** pair

ZOLTAN_NUM_GEOM_FN
ZOLTAN_GEOM_MULTI_FN or **ZOLTAN_GEOM_FN**

C:	int **Zoltan_RCB_Box** (struct **Zoltan_Struct** * *zz*, int *part*, int **ndim*, double **xmin*, double **ymin*, double **zmin*, double **xmax*, double **ymax*, double **zmax*);
FORTRAN:	FUNCTION **Zoltan_RCB_Box**(*zz*, *part*,*ndim*, *xmin*, *ymin*, *zmin*, *xmax*, *ymax*, *zmax*) INTEGER(Zoltan_INT) :: Zoltan_RCB_Box TYPE(Zoltan_Struct), INTENT(IN) :: zz INTEGER(Zoltan_INT), INTENT(IN) :: part INTEGER(Zoltan_INT), INTENT(OUT) :: ndim REAL(Zoltan_DOUBLE), INTENT(OUT) :: xmin, ymin, zmin, xmax, ymax, zmax

---

In many settings, it is useful to know a partition's bounding box generated by RCB. This bounding box describes the region of space assigned to a given partition. Given an RCB decomposition of space and a partition number, **Zoltan_RCB_Box** returns the lower and upper corners of the region of space assigned to the partition. To use this routine, the parameter **KEEP_CUTS** must be set to TRUE when the decomposition is generated. This parameter will cause the sequence of geometric cuts to be saved, which is necessary for **Zoltan_RCB_Box** to do its job.

Arguments:	
zz	Pointer to the Zoltan structure created by **Zoltan_Create**.
part	Partition number of partition for which the bounding box should be returned.
ndim	Upon return, the number of dimensions in the partitioned geometry.
xmin, *ymin*, *zmin*	Upon return, the coordinates of the lower extent of bounding box for the partition. If the geometry is two-dimensional, *zmin* is -DBL_MAX. If the geometry is one-dimensional, *ymin* is -DBL_MAX.
xmax, *ymax*, *zmax*	Upon return, the coordinates of the upper extent of bounding box for the partition. If the geometry is two-dimensional, *zmax* is DBL_MAX. If the geometry is one-dimensional, *ymax* is DBL_MAX.
Returned Value:	
int	Error code.

[Table of Contents | Next: Recursive Inertial Bisection (RIB) | Previous: Load-Balancing Algorithms]

Recursive Inertial Bisection (RIB)

An implementation of Recursive Inertial Bisection (RIB) is included in Zoltan. RIB was proposed as a load-balancing algorithm by Williams and later studied by Taylor and Nour-Omid, but its origin is unclear. RIB is similar to RCB in that it divides the domain based on the location of the objects being partitioned by use of cutting planes. In RIB, the computational domain is first divided into two regions by a cutting plane orthogonal to the longest direction of the domain so that half the work load is in each of the sub-regions. The sub-regions are then further divided by recursive application of the same splitting algorithm until the number of sub-regions equals the number of processors. Although this algorithm was first devised to cut into a number of sets which is a power of two, the set sizes in a particular cut needn't be equal. By adjusting the partition sizes appropriately, any number of equally-sized sets can be created. If the parallel machine has processors with different speeds, sets with nonuniform sizes can also be easily generated. The Zoltan implementation of RIB has several parameters which can be modified by the **Zoltan_Set_Param** function.

Method String:	**RIB**
Parameters:	
RIB_OVERALLOC	The amount by which to over-allocate temporary storage arrays for objects within the RIB algorithm when additional storage is due to changes in processor assignments. 1.0 = no extra storage allocated; 1.5 = 50% extra storage; etc.
RIB_OUTPUT_LEVEL	Flag controlling the amount of timing and diagnostic output the routine produces. 0 = no output; 1 = print summary; 2 = print data for each processor.
CHECK_GEOM	Flag controlling the invocation of input and output error checking. 0 = don't do checking; 1 = do checking.
KEEP_CUTS	Should information about the cuts determining the RIB decomposition be retained? It costs a bit of time to do so, but this information is necessary if application wants to add more objects to the decomposition via calls to **Zoltan_LB_Point_PP_Assign** or to **Zoltan_LB_Box_PP_Assign**. 0 = don't keep cuts; 1 = keep cuts.
AVERAGE_CUTS	When set to one, coordinates of RIB cutting planes are computed to be the average of the coordinates of the closest object on each side of the cut. Otherwise, coordinates of cutting planes may equal those of one of the closest objects. 0 = don't average cuts; 1 = average cuts.

REDUCE_DIMENSIONS	When a 3 dimensional geometry is almost flat, it may make more sense to treat it as a 2 dimensional geometry when applying the RIB algorithm. (Coordinate values in the omitted direction are ignored for the purposes of partitioning.) If this parameter is set to **1**, a 3 dimensional geometry will be treated as 2 dimensional if it is very flat, or 1 dimensional if it is very thin. A 2 dimensional geometry will be treated as 1 dimensional if it is very thin.
DEGENERATE_RATIO	If the **REDUCE_DIMENSIONS** parameter is set, then this parameter determines when a geometry is considered to be degenerate. A bounding box which is oriented to the geometry is constructed, and the lengths of its sides are tested against a ratio of 1 : **DEGENERATE_RATIO**.

Default:

RIB_OVERALLOC = 1.0
RIB_OUTPUT_LEVEL = 0
CHECK_GEOM = 1
KEEP_CUTS = 0
AVERAGE_CUTS = 0
REDUCE_DIMENSIONS = 0
DEGENERATE_RATIO = 10

Required Query Functions:

ZOLTAN_NUM_OBJ_FN
ZOLTAN_OBJ_LIST_FN or
ZOLTAN_FIRST_OBJ_FN/**ZOLTAN_NEXT_OBJ_FN** pair
ZOLTAN_NUM_GEOM_FN
ZOLTAN_GEOM_MULTI_FN or **ZOLTAN_GEOM_FN**

[Table of Contents | Next: Hilbert Space-Filling Curve Partitioning | Previous: Recursive Coordinate Bisection (RCB)]

Hilbert Space Filling Curve (HSFC)

The Inverse Hilbert Space-Filling Curve functions map a point in one, two or three dimensions into the interval [0,1]. The Hilbert functions that map [0, 1] to normal spatial coordinates are also provided. (The one-dimensional inverse Hilbert curve is defined here as the identity function, f(x)=x for all x.)

The HSFC partitioning algorithm seeks to divide [0,1] into P intervals each containing the same weight of objects associated to these intervals by their inverse Hilbert coordinates. N bins are created (where N > P) to partition [0,1]. The weights in each bin are summed across all processors. A greedy algorithm sums the bins (from left to right) placing a cut when the desired weight for current partition interval is achieved. This process is repeated as needed to improve partitioning tolerance by a technique that maintains the same total number of bins but refines the bins previously containing a cut.

HSFC returns an warning if the final imbalance exceeds the user specified tolerance.

This code implements both the point assign and box assign functionality. The point assign determines an appropriate partition (associated with a specific group of processors) for a new point. The box assign determines the list of processors whose associated subdomains intersect the given box. In order to use either of these routines, the user parameter KEEP_CUTS must be turned on. Both point assign and box assign now work for points or boxes anywhere in space even if they are exterior to the original bounding box. If a partition is empty (due to the partition being assigned zero work), it is not included in the list of partitions returned by box assign. Note: the original box assign algorithm was not rigorous and may have missed partitions. This version is both rigorous and fast.

The Zoltan implementation of HSFC has one parameter that can be modified by the **Zoltan_Set_Param** function.

This partitioning algorithm is loosely based on the 2D & 3D Hilbert tables used in the Octree partitioner and on the BSFC partitioning implementation by Andrew C. Bauer, Department of Engineering, State University of New York at Buffalo, as his summer project at SNL in 2001. The box assign algorithm is loosely based on the papers by Lawder referenced both in the developers guide and the code itself. NOTE: This code can be trivially extended to any space filling curve by providing the tables implementing the curve's state transition diagram. The only dependance on the curve is through the tables and the box assign algorithm will work for all space filling curves (if we have their tables.)

Please refer to the Zoltan Developers Guide, **Appendix: Hilbert Space Filling Curve (HSFC)** for more detailed information about these algorithms.

Method String:	**HSFC**
Parameters:	

KEEP_CUTS	Information about cuts and bounding box is necessary if the application wants to add more objects to the decomposition via calls to **Zoltan_LB_Point_PP_Assign** or to **Zoltan_LB_Box_PP_Assign**. 0 = don't keep cuts; 1 = keep cuts.
REDUCE_DIMENSIONS	When a 3 dimensional geometry is almost flat, it may make more sense to treat it as a 2 dimensional geometry when applying the HSFC algorithm. (Coordinate values in the omitted direction are ignored for the purposes of partitioning.) If this parameter is set to **1**, a 3 dimensional geometry will be treated as 2 dimensional if is very flat, or 1 dimensional if it very thin. And a 2 dimensional geometry will be treated as 1 dimensional if it is very thin. Turning this parameter on removes the possibility that disconnected partitions will appear on the surface of a flat 3 dimensional object.
DEGENERATE_RATIO	If the **REDUCE_DIMENSIONS** parameter is set, then this parameter determines when a geometry is considered to be flat. A bounding box which is oriented to the geometry is constructed, and the lengths of it's sides are tested against a ratio of 1 : **DEGENERATE_RATIO**.

Default:

KEEP_CUTS = 0
REDUCE_DIMENSIONS = 0
DEGENERATE_RATIO = 10

Required Query Functions:

ZOLTAN_NUM_OBJ_FN
ZOLTAN_OBJ_LIST_FN or **ZOLTAN_FIRST_OBJ_FN**/**ZOLTAN_NEXT_OBJ_FN** pair
ZOLTAN_NUM_GEOM_FN
ZOLTAN_GEOM_MULTI_FN or **ZOLTAN_GEOM_FN**

[Table of Contents | Next: Refinement Tree Partitioning | Previous: Recursive Inertial Bisection]

Refinement Tree Partitioning (REFTREE)

The refinement tree based partitioning method is due to William Mitchell of the National Institute of Standards and Technology [Mitchell]. It is closely related to the Octree and Space-Filling Curve methods, except it uses the tree that represents the adaptive refinement process that created the grid. This tree is constructed through the tree-based query functions.

Each node of the refinement tree corresponds to an element that occurred during the grid refinement process. The first level of the tree (the children of the root of the tree) corresponds to the initial coarse grid, one tree node per initial element. It is assumed that the initial coarse grid does not change through the execution of the program, except that the local IDs, assignment of elements to processors, and weights can change. If any other aspect of the coarse grid changes, then the Zoltan structure should be destroyed and recreated. The children of a node in the tree correspond to the elements that were created when the corresponding element was refined. The children are ordered such that a traversal of the tree creates a space-filling curve within each initial element. If the initial elements can be ordered with a contiguous path through them, then the traversal creates a space-filling curve through all the elements. Each element has a designated "in" vertex and "out" vertex, with the out vertex of one element being the same as the in vertex of the next element in the path, in other words the path goes through a vertex to move from one element to the next (and does not go out the same vertex it came in).

The user may allow Zoltan to determine the order of the coarse grid elements, or may specify the order, which might be faster or produce a better path. If Zoltan determines the order, the user can select between an order that will produce connected partitions, an order based on a Hilbert Space Filling Curve, or an order based on a Sierpinski Space Filling Curve. See the parameter REFTREE_INITPATH below. If the user provides the order, then the in/out vertices must also be supplied. Similarly, the user may specify the order and in/out vertices of the child elements, or allow Zoltan to determine them. If the user knows how to provide a good ordering for the children, this may be significantly faster than the default general algorithm. However, accelerated forms of the ordering algorithm are provided for certain types of refinement schemes and should be used in those cases. See **ZOLTAN_CHILD_LIST_FN**. If the user always specifies the order, then the vertices and in/out vertices are not used and do not have to be provided.

Weights are assigned to the nodes of the tree. These weights need not be only on the leaves (the elements of the final grid), but can also be on interior nodes (for example, to represent work on coarse grids of a multigrid algorithm). The default weights are 1.0 at the leaves and 0.0 at the interior nodes, which produces a partition based on the number of elements in each partition. An initial tree traversal is used to sum the weights, and a second traversal to cut the space-filling curve into appropriately-sized pieces and assign elements to partitions. The number of partitions is not necessarily equal to the number of processors.

The following limitations should be removed in the future.

- For multicomponent weights, only the first component is used.
- Heterogeneous architectures are not supported, in the sense that the computational load is equally divided over the processors. A vector of relative partition sizes is used to determine the weight assigned to each partition, but they are currently all equal. In the future they should be input to reflect heterogeneity.

Method String:	**REFTREE**
Parameters:	
REFTREE_HASH_SIZE	The size of the hash table to map from global IDs to refinement tree nodes. Larger values require more memory but may reduce search time.
Default:	
	REFTREE_HASH_SIZE = 16384
REFTREE_INITPATH	Determines the method for finding an order of the elements in the initial grid. "SIERPINSKI" uses a Sierpinski Space Filling Curve and is most appropriate for grids consisting of triangles. It is currently limited to 2D. "HILBERT" uses a Hilbert Space Filling Curve and is most appropriate for grids consisting of quadralaterals or hexahedra. "CONNECTED" attempts to produce connected partitions (guaranteed for triangles and tetrahedra), however they tend to be stringy, i.e., less compact than the SFC methods. It is most appropriate when connected partitions are required. An invalid character string will invoke the default method.
Default:	
	REFTREE_INITPATH = "SIERPINSKI" if the grid contains only triangles *REFTREE_INITPATH* = "HILBERT" otherwise *NOTE:* In Zoltan versions 1.53 and earlier the default was "CONNECTED". To reproduce old results, use *REFTREE_INITPATH* = "CONNECTED".
Required Query Functions:	
	ZOLTAN_NUM_COARSE_OBJ_FN **ZOLTAN_COARSE_OBJ_LIST_FN** or **ZOLTAN_FIRST_COARSE_OBJ_FN**/**ZOLTAN_NEXT_COARSE_OBJ_FN** pair **ZOLTAN_NUM_CHILD_FN** **ZOLTAN_CHILD_LIST_FN** **ZOLTAN_CHILD_WEIGHT_FN** The following functions are needed only if the order of the initial elements will be determined by a space filling curve method: **ZOLTAN_NUM_GEOM_FN** **ZOLTAN_GEOM_FN** or **ZOLTAN_GEOM_MULTI_FN**

[Table of Contents | Next: ParMETIS | Previous: Hilbert Space-Filling Curve Partitioning]

Note: See also hypergraph partitioning.

Graph partitioning: ParMETIS

ParMETIS is a parallel library for graph partitioning (for static load balancing) and repartitioning (for dynamic load balancing) developed at the University of Minnesota by Karypis, Schloegel and Kumar [Karypis and Kumar]. ParMETIS is therefore strictly speaking not a method but rather a collection of methods. In the Zoltan context, ParMETIS is a method with many sub-methods. Zoltan provides an interface to all the ParMETIS (sub-)methods. The user selects which ParMETIS method to use through the parameter PARMETIS_METHOD. Most of the ParMETIS methods are based on either multilevel Kernighan-Lin partitioning or a diffusion algorithm. The names of the ParMETIS methods used by Zoltan are identical to those in the ParMETIS library. For further information about the various ParMETIS methods and parameters, please consult the ParMETIS User's Guide.

Graph partitioning is a useful abstraction for load balancing. The main idea is to represent the computational application as a weighted graph. The nodes or vertices in the graph correspond to objects in Zoltan. Each object may have a weight that normally represents the amount of computation. The edges or arcs in the graph usually correspond to communication costs. In graph partitioning, the problem is to find a partitioning of the graph (that is, each vertex is assigned to one out of *k* possible sets called partitions) that minimizes the cut size (weight) subject to the partitions having approximately equal size (weight). In repartitioning, it is assumed that a partitioning already exists. The problem is to find a good partitioning that is also "similar" in some sense to the existing partitioning. This keeps the migration cost low. All the problems described above are NP-hard so no efficient exact algorithm is known. We remark that in Zoltan 1.*, the number of partitions is always the same as the number of MPI processes (which is normally equal to the number of processors).

We give only a brief summary of the various ParMETIS methods here; for more details see the ParMETIS documentation. The methods fall into three categories:

1. Part* - Perform graph partitioning without consideration of the initial distribution.
2. AdaptiveRepart (ParMETIS 3) and Repart* (ParMETIS 2) - Incremental algorithms with small migration cost.
3. Refine* - Refines a given partitioning (balance). Can be applied multiple times to reduce the communication cost (cut weight) if desired.

As a rule of thumb, use one of the Part* methods if you have a poor initial balance and you are willing to spend some time doing migration. One such case is static load balancing; that is, you need to balance only once. Use AdaptiveRepart or the Repart* methods when you have a reasonably good load balance that you wish to update incrementally. These methods are well suited for dynamic load balancing (for example, adaptive mesh refinement). A reasonable strategy is to call PartKway once to obtain a good initial balance and later update this balance using AdaptiveRepart (Repart* in ParMetis 2.0).

Zoltan is currently compatible with ParMETIS versions 3.1 and 2.0. There is no guarantee that Zoltan will work correctly if you have a different version of ParMETIS on your computer. (ParMETIS 3.0 will work with Zoltan in most cases, but is not officially supported. ParMETIS 3.1 is highly recommended.

The 2.0 version will soon become obsolete and may not be supported in future Zoltan versions.) The ParMETIS source code can be obtained from the [ParMETIS home page](). As a courtesy service, a recent, compatible version of the ParMETIS source code is distributed with Zoltan. However, ParMETIS is a completely separate library. If you do not wish to install ParMETIS, it is possible to compile Zoltan without any references to ParMETIS (when you 'make' Zoltan, comment out the PARMETIS_LIBPATH variable in the configuration file *Utilities/Config/Config.<platform>*).

Note that Zoltan ignores the imbalance tolerance parameter IMBALANCE_TOL when ParMETIS 2.0 is used (the default value 1.05 is used), while IMBALANCE_TOL works correctly with ParMETIS 3.0. Zoltan supports the multiconstraint feature of ParMETIS 3 through multiple object weights (see *OBJ_WEIGHT_DIM*).

The graph given to Zoltan/ParMETIS must be symmetric. Any self edges (loops) will be ignored. Multiple edges between a pair of vertices is not allowed. All weights must be non-negative. The graph does not have to be connected.

Method String:	**GRAPH** or **PARMETIS**
Parameters:	
PARMETIS_METHOD	The ParMETIS method to be used; currently nine are available. *PartKway* - multilevel Kernighan-Lin partitioning *PartGeom* - space filling curves (coordinate based) *PartGeomKway* - hybrid method based on PartKway and PartGeom (needs both graph data and coordinates) *AdaptiveRepart* - adaptive repartitioning (only in ParMETIS 3.0 and higher) *RepartLDiffusion* - diffusion algorithm (local) *RepartGDiffusion* - diffusion algorithm (global) *RepartRemap* - multilevel partioning with remap seeking to minimize migration cost *RepartMLRemap* - similar to RepartRemap but with additional multilevel refinement *RefineKway* - refine the current partitioning (balance) The method names are case insensitive.
PARMETIS_OUTPUT_LEVEL	Amount of output the load-balancing algorithm should produce. 0 = no output, 1 = print timing info. Turning on more bits displays more information (for example, 3=1+2, 5=1+4, 7=1+2+4).
PARMETIS_COARSE_ALG	Coarse algorithm for PartKway. 1 = serial, 2 = parallel. (ParMETIS 2 only)
PARMETIS_SEED	Random seed for ParMETIS.
PARMETIS_ITR	Ratio of interprocessor communication time to redistribution time. A high value will emphasize reducing the edge cut, while a small value will try to keep the change in the new partition (distribution) small. This parameter is only used by AdaptiveRepart. A value of between 100 and 1000 is good for most problems.

USE_OBJ_SIZE	Use (or not use) the available information about object sizes to estimate migration cost. This parameter is currently only relevant for AdaptiveRepart.
CHECK_GRAPH	Level of error checking for graph input: 0 = no checking, 1 = on-processor checking, 2 = full checking. (CHECK_GRAPH==2 is very slow and should be used only during debugging).
SCATTER_GRAPH	Scatter graph data by distributing contiguous chunks of objects (vertices) of roughly equal size to each processor before calling the partitioner. 0 = don't scatter; 1 = scatter only if all objects are on a single processor; 2 = scatter if at least one processor owns no objects (recommended to avoid a bug in ParMETIS 2.0); 3 = always scatter.
Default values:	
	PARMETIS_METHOD = RepartGDiffusion *PARMETIS_OUTPUT_LEVEL* = 0 *PARMETIS_COARSE_ALG* = 2 *PARMETIS_SEED* = 15 *PARMETIS_ITR* = 100 *USE_OBJ_SIZE* = 1 *CHECK_GRAPH* = 1 *SCATTER_GRAPH* = 1
Required Query Functions:	
For all submethods:	**ZOLTAN_NUM_OBJ_FN** **ZOLTAN_OBJ_LIST_FN** or **ZOLTAN_FIRST_OBJ_FN**/**ZOLTAN_NEXT_OBJ_FN** pair
Only PartGeom & PartGeomKway:	**ZOLTAN_NUM_GEOM_FN** **ZOLTAN_GEOM_MULTI_FN** or **ZOLTAN_GEOM_FN**
All but PartGeom:	**ZOLTAN_NUM_EDGES_MULTI_FN** or **ZOLTAN_NUM_EDGES_FN** **ZOLTAN_EDGE_LIST_MULTI_FN** or **ZOLTAN_EDGE_LIST_FN**

[Table of Contents | Next: Jostle | Previous: Refinement Tree Partitioning]

Graph partitioning: Jostle

Jostle is a library for graph (mesh) partitioning and load balancing developed at the University of Greenwich, London, UK, by Chris Walshaw [Jostle, Walshaw]. The parallel version of Jostle is sometimes called pjostle. In the Zoltan context, the name Jostle always refers to the parallel version of the library. The main algorithm used in Jostle is based on multilevel graph partitioning, and a diffusion-type method is available for repartitioning. Hence the Jostle library is very similar to ParMETIS. See the ParMETIS section for a brief description of graph partitioning as a model for load balancing.

At present, only the most common Jostle options are supported by Zoltan. These are briefly described below. For further details, see the documentation available from the Jostle home page. Other options may be added to Zoltan upon request.

Note that Jostle is not distributed with Zoltan. If you wish to use Jostle within Zoltan, you must first obtain a license for Parallel Jostle and install it on your system. The license is currently free for academic use. Zoltan has been tested only with parallel Jostle version 1.2.* and may be incompatible with other versions. Zoltan offers only limited support for Jostle and this may be discontinued in the future.

Method String:	**JOSTLE**
Parameters:	
JOSTLE_OUTPUT_LEVEL	Amount of output Jostle should produce. (integer)
JOSTLE_THRESHOLD	Threshold at which the graph contraction phase is stopped. (integer)
JOSTLE_GATHER_THRESHOLD	Duplicate coarse graph on all processors when there are fewer than this number of nodes. (integer)
JOSTLE_MATCHING	Matching algorithm for graph contraction. (Valid values are "local" and "global".)
JOSTLE_REDUCTION	When reduction is turned off, Jostle performs a diffusion-type algorithm instead of multilevel graph partitioning. (Valid values are "on" and "off".)
JOSTLE_CONNECT	Make a disconnected graph connected before partitioning. (Valid values are "on" and "off".)
CHECK_GRAPH	Level of error checking for graph input: 0 = no checking, 1 = on-processor checking, 2 = full checking. (CHECK_GRAPH==2 is very slow and should be used only during debugging).
SCATTER_GRAPH	Scatter graph data by distributing contiguous chunks of objects (vertices) of roughly equal size to each processor before calling the partitioner. 0 = don't scatter; 1 = scatter only if all objects are on a single processor; 2 = scatter if at least one processor owns no objects; 3 = always scatter.
Default values:	See the Jostle documentation. See our ParMETIS section for the last two parameters.

Required Query Functions:

ZOLTAN_NUM_OBJ_FN
ZOLTAN_OBJ_LIST_FN or
ZOLTAN_FIRST_OBJ_FN/**ZOLTAN_NEXT_OBJ_FN** pair
ZOLTAN_NUM_EDGES_MULTI_FN or
ZOLTAN_NUM_EDGES_FN
ZOLTAN_EDGE_LIST_MULTI_FN or
ZOLTAN_EDGE_LIST_FN

[Table of Contents | Next: Hypergraph Partitioning | Previous: ParMETIS]

Hypergraph Partitioning

Hypergraph partitioning is a useful partitioning and load balancing method when connectivity data is available. It can be viewed as a more sophisticated alternative to the traditional graph partitioning.

A hypergraph consists of vertices and hyperedges. A hyperedge connects one or more vertices. A graph is a special case of a hypergraph where each edge has size two (two vertices). The hypergraph model is well suited to parallel computing, where vertices correspond to data objects and hyperedges represent the communication requirements. The basic partitioning problem is to partition the vertices into *k* approximately equal sets such that the number of cut hyperedges is minimized. Most partitioners (including Zoltan-PHG) allows a more general model where both vertices and hyperedges can be assigned weights. It has been shown that the hypergraph model gives a more accurate representation of communication cost (volume) than the graph model. In particular, for sparse matrix-vector multiplication, the hypergraph model **exactly** represents communication volume. Sparse matrices can be partitioned either along rows or columns; in the row-net model the columns are vertices and each row corresponds to an hyperedge, while in the column-net model the roles of vertices and hyperedges are reversed.

Zoltan contains a native parallel hypergraph partitioner, called PHG (Parallel HyperGraph partitioner). In addition, Zoltan provides access to PaToH, a serial hypergraph partitioner. Note that PaToH is not part of Zoltan and should be obtained separately from the PaToH web site. Zoltan-PHG is a fully parallel multilevel hypergraph partitioner. For further technical description, see [Devine et al, 2006].

Planned future features (currently not supported):

- Fixed vertices
- Repartitioning (to reduce migration cost)
- Multiconstraint partitioning
- 2-dimensional (sparse matrix) partitioning

For applications that already use Zoltan to do graph partitioning, it is fairly easy to upgrade to hypergraph partitioning. See the section graph vs. hypergraph partitioning.

Method String:	**HYPERGRAPH**
Parameters:	
HYPERGRAPH_PACKAGE	PHG (parallel) or PaToH (serial)
PHG_OUTPUT_LEVEL	Level of verbosity; 0 is silent.
CHECK_HYPERGRAPH (CHECK_GRAPH)	Check that the query functions return valid input data; 0 or 1. (This slows performance; intended for debugging.)

PHG_COARSENING_METHOD	The method to use in the matching/coarsening phase: ipm - inner product matching (a.k.a. heavy connectivity matching); gives best quality. l-ipm - local ipm on each processor. Faster but usually gives poorer quality. a-ipm - alternate between ipm and l-ipm. (A compromise between speed and quality.) none - no coarsening
PHG_COARSEPARTITION_METHOD	Method to partition the coarsest (smallest) hypergraph; typically done in serial: random - random linear - linear assignment of the vertices (ordered by the user query function) greedy - greedy method based on minimizing cuts auto - automatically select from the above methods (in parallel, the processes will do different methods)
PHG_REFINEMENT_METHOD	Refinement algorithm: fm - approximate Fiduccia-Mattheyses (FM) no - no refinement
PHG_REFINEMENT_QUALITY	Knob to control the trade-off between run time and quality. 1 is the recommended (default) setting, >1 gives more refinement (higher quality partitions but longer run time), while <1 gives less refinement (and poorer quality).
PHG_RANDOMIZE_INPUT	Randomize layout of vertices and hyperedges in internal parallel 2D layout? Setting this parameter to 1 often reduces Zoltan-PHG execution time.
PHG_EDGE_WEIGHT_OPERATION	Operation to be applied to edge weights supplied by different processes for the same hyperedge: *add* - the hyperedge weight will be the sum of the supplied weights *max* - the hyperedge weight will be the maximum of the supplied weights *error* - if the hyperedge weights are not equal, Zoltan will flag an error, otherwise the hyperedge weight will be the value returned by the processes
PHG_EDGE_SIZE_THRESHOLD	Ignore hyperedges greater than this fraction times number of vertices.
ADD_OBJ_WEIGHT	Add implicit vertex (object) weight. This will be in addition to the user-defined weights. (Note: Multi-weight partitioning is not yet supported, so currently only use this option with obj_weight_dim=0.) Valid values: "none" "unit" or "vertices" "pins" or "nonzeros" or "vertex degree"

Default values:

HYPERGRAPH_PACKAGE = PHG
CHECK_HYPERGRAPH=0
PHG_OUTPUT_LEVEL=0
PHG_COARSENING_METHOD=ipm
PHG_COARSEPARTITION_METHOD=auto
PHG_REFINEMENT_METHOD=fm
PHG_REFINEMENT_QUALITY=1
PHG_RANDOMIZE_INPUT=0
PHG_EDGE_WEIGHT_OPERATION=max
PHG_EDGE_SIZE_THRESHOLD=0.25
ADD_OBJ_WEIGHT=none

Required Query Functions:

ZOLTAN_NUM_OBJ_FN
ZOLTAN_OBJ_LIST_FN or **ZOLTAN_FIRST_OBJ_FN**/**ZOLTAN_NEXT_OBJ_FN** pair
ZOLTAN_HG_SIZE_CS_FN
ZOLTAN_HG_CS_FN

Optional Query Functions:

ZOLTAN_HG_SIZE_EDGE_WTS_FN
ZOLTAN_HG_EDGE_WTS_FN

It is possible to provide the graph query functions instead of the hypergraph queries, though this is not recommended. If only graph query functions are registered, Zoltan will automatically create a hypergraph from the graph, but this is not equivalent to graph partitioning. In particular, the edge weights will not be accurate.

[Table of Contents | Next: Octree | Previous: Jostle]

Octree Partitioning (OCTPART)

The Octree Partitioning algorithm is based upon work in load balancing for parallel mesh generation at Rensselaer Polytechnic Institute [Flaherty, Loy et al.]. It was implemented in Zoltan by Luis Gervasio, Department of Computer Science, Rensselaer Polytechnic Institute, as his summer project in 1998 [Gervasio]. An octree is a spatial decomposition of the computational domain in which the root of the tree, representing the entire domain, is recursively divided by two in each coordinate direction (producing eight or four "child" octants in 3D or 2D, respectively) until each subregion holds at most an application-specified number of objects. These subregions are represented by the leaves of the octree. The octree data structure is widely used in mesh generation and adaptive mesh refinement [Baehmann et al., Shephard and Georges]. The octree resulting from such a spatial decomposition of the domain can be used to partition an application's work [Edwards, Pilkington and Baden, Warren and Salmon]. To partition an octree, a traversal of the tree is used to define a global ordering on the leaves of the octree. This global ordering is often referred to as a Space-Filling Curve (SFC). The leaves of the octree can be easily assigned to processors in a manner which equally distributes work by assigning slices of the ordered list to processors. Different tree-traversal algorithms produce different global orderings or SFCs, with some SFCs having better connectivity and partition quality properties than others. Currently, Morton Indexing (i.e., Z-curve), Grey Code, and Hilbert SFCs are supported. Morton Indexing and Grey Code SFCs are the simplest (and currently, the fastest) of the SFC algorithms, but they produce lower-quality partitions than the Hilbert SFC.

Method String:	**OCTPART**
Parameters:	
OCT_DIM	Specifies whether the 2D or 3D Octree algorithms should be used. The 3D algorithms can be used for 2D problems, but much memory will be wasted to allow for a non-existent third dimension. Similarly, a 2D algorithm can be used for 3D surface meshes provided that the surface can be projected to the *xy*-plane without overlapping points. 2 = use 2D algorithm; 3 = use 3D algorithm.
OCT_METHOD	The SFC to be used. 0 = Morton Indexing; 1 = Grey Code; 2 = Hilbert.
OCT_MINOBJECTS	The minimum number of objects to allow in a leaf octant of the octree. These objects will be assigned as a group to a processor, so this parameter helps define the granularity of the load-balancing problem. Values greater than or equal to one are allowable.
OCT_MAXOBJECTS	The maximum number of objects to allow in a leaf octant of the octree. These objects will be assigned as a group to a processor, so this parameter helps define the granularity of the load-balancing problem. Values greater than or equal to one are allowable.

OCT_OUTPUT_LEVEL	Amount of output the load-balancing algorithm should produce. 0 = no statistics; 1 = statistics summary; 2 = debugging information; 3 = data for generating plots.

Default:

OCT_DIM = 3
OCT_METHOD = 2
OCT_MINOBJECTS = 10
OCT_MAXOBJECTS = 40
OCT_OUTPUT_LEVEL = 0

Required Query Functions:

ZOLTAN_NUM_OBJ_FN
ZOLTAN_OBJ_LIST_FN or **ZOLTAN_FIRST_OBJ_FN**/**ZOLTAN_NEXT_OBJ_FN** pair
ZOLTAN_NUM_GEOM_FN
ZOLTAN_GEOM_MULTI_FN or **ZOLTAN_GEOM_FN**

[Table of Contents | Next: Ordering | Previous: Hypergraph Partitioning]

Ordering Algorithms

The following graph ordering algorithms are currently included in the Zoltan library:

Nested dissection by METIS/ParMETIS (NODEND)

The parenthetical string is the parameter value for *ORDER_METHOD* parameter; the parameter is set through a call to **Zoltan_Set_Param**.

Ordering Parameters

While the overall behavior of Zoltan is controlled by general Zoltan parameters, the behavior of each ordering method is controlled by parameters specific to ordering which are also set by calls to **Zoltan_Set_Param**. Many of these parameters are specific to individual ordering algorithms, and are listed in the descriptions of the individual algorithms. However, several have meaning across multiple ordering algorithms. These parameters are described below.

Parameters:	
ORDER_METHOD	The order algorithm used by Zoltan is specified by this parameter. Valid values are "NODEND" (for nodal nested dissection by ParMETIS or METIS), "METIS" (same as NODEND with ORDER_TYPE = local), "PARMETIS" (same as NODEND with ORDER_TYPE = global), and "NONE" (for no load-balancing).
ORDER_TYPE	"LOCAL" or "GLOBAL". If LOCAL is selected, then each processor constructs a local (sub-)graph. All inter-processor edges are simply ignored. The ordering arrays returned, *rank* and *iperm,* are local permutation vectors in this case.
ORDER_START_INDEX	The start index for the permutation vectors *rank* and *iperm*. Valid values are 0 and 1.
REORDER	If this value is set to TRUE (non-zero), Zoltan assumes that the lists of local and global ids are given as input to **Zoltan_Order.** Otherwise, the id lists will be populated by **Zoltan_Order**. The permutation of the ids will be the one produced by calling the query functions.
Default Values:	
	ORDER_METHOD = NODEND *ORDER_TYPE* = GLOBAL *ORDER_START_INDEX* = 0 *REORDER* = FALSE

[Table of Contents | Next: Nested dissection by ParMETIS | Previous: Octtree Algorithm]

Nested Dissection by METIS/ParMETIS

Nested Dissection (ND) is a popular method to compute fill-reducing orderings for sparse matrices. It can also be used for other ordering purposes. The algorithm recursively finds a separator (bisector) in a graph, orders the nodes in the two subsets first, and nodes in the separator last. In METIS/ParMETIS, the recursion is stopped when the graph is smaller than a certain size, and some version of minimum degree ordering is applied to the remaining graph.

METIS computes a local ordering of the objects on each processor, while ParMETIS performs a global ordering of all the objects. ParMETIS currently (versions 2.0 and 3.0) requires that the number of processors is a power of two.

The generic name for this method is NODEND. If GRAPH_TYPE=GLOBAL ParMETIS is called, but if it is LOCAL, METIS is called. Alternatively, the user can simply set ORDER_METHOD to METIS or PARMETIS.

Order_Method String:	**NODEND or METIS or PARMETIS**
Parameters:	
See ParMETIS.	Note that the PARMETIS options are ignored when METIS is called.
Required Query Functions:	
	ZOLTAN_NUM_OBJ_FN **ZOLTAN_OBJ_LIST_FN** or **ZOLTAN_FIRST_OBJ_FN**/**ZOLTAN_NEXT_OBJ_FN** pair **ZOLTAN_NUM_EDGES_MULTI_F N** or **ZOLTAN_NUM_EDGES_FN** **ZOLTAN_EDGE_LIST_MULTI_F N** or **ZOLTAN_EDGE_LIST_FN**

Zoltan User's Guide | ***Next*** | ***Previous***

Coloring Algorithms

The following coloring algorithms are currently included in the Zoltan library:

Parallel Coloring

Coloring Parameters

While the overall behavior of Zoltan is controlled by general Zoltan parameters, the behavior of each coloring method is controlled by parameters specific to coloring which are also set by calls to **Zoltan_Set_Param**. These parameters are described below.

Parameters:

DISTANCE	The maximum distance between two objects that should not get the same color is specified by this parameter. Valid values are "1" (for distance-1 coloring) and "2" (for distance-2 coloring).
SUPERSTEP_SIZE	Number of local objects to be colored on each processor before exchanging color information. SUPERSTEP_SIZE should be greater than 0.
COMM_PATTERN	Valid values are "S" (synchronous) and "A" (asynchronous). If synchronous communication is selected, processors are forced to wait for the color information from all other processors to be received before proceeding with coloring of the next SUPERSTEP_SIZE number of local objects. If asynchronous communication is selected, there is no such restriction.
COLOR_ORDER	Valid values are "I" (internal first), "B" (boundary first) and "U" (unordered). If "I" is selected, each processor colors its internal objects before boundary objects. If "B" is selected, each processor colors its boundary objects first. If "U" is selected, there is no such distinction between internal and boundary objects. "U" is not implemented for distance-2 coloring.
COLORING_METHOD	Currently only "F" (first-fit) is implemented. By using "F", the smallest available color that will not cause a conflict is assigned to the object that is being colored.

Default Values:

DISTANCE = 1
SUPERSTEP_SIZE = 100
COMM_PATTERN = S
COLOR_ORDER = I
COLORING_METHOD = F

[Table of Contents | Next: Parallel Coloring | Previous: Nested Dissection by ParMETIS]

Parallel Coloring

The parallel coloring algorithm in Zoltan is based on the work of Boman et al. for distance-1 coloring and Bozdag et al. for distance-2 coloring. It was implemented in Zoltan by Doruk Bozdag and Umit Catalyurek, Department of Biomedical Informatics, Ohio State University. Distance-1 coloring algorithm is an iterative data parallel algorithm that proceeds in two-phased rounds. In the first phase, processors concurrently color the vertices assigned to them. Adjacent vertices colored in the same parallel step of this phase may result in inconsistencies. In the second phase, processors concurrently check the validity of the colors assigned to their respective vertices and identify a set of vertices that needs to be re-colored in the next round to resolve the detected inconsistencies. The algorithm terminates when every vertex has been colored correctly. To reduce communication frequency, the coloring phase is further decomposed into computation and communication sub-phases. In a communication sub-phase processors exchange recent color information. During a computation sub-phase, a number of vertices determined by the SUPERSTEP_SIZE parameter, rather than a single vertex, is colored based on currently available color information. With an appropriate choice of a value for SUPERSTEP_SIZE, the number of ensuing conflicts can be kept low while at the same time preventing the runtime from being dominated by the sending of a large number of small messages. The distance-2 graph coloring problem aims at partitioning the vertex set of a graph into the fewest sets consisting of vertices pairwise at distance greater than two from each other. The algorithm is an extension of the parallel distance-1 coloring algorithm.

Parameters:
See Coloring Algorithms.

Required Query Functions:

ZOLTAN_NUM_OBJ_FN
ZOLTAN_OBJ_LIST_FN or **ZOLTAN_FIRST_OBJ_FN**/**ZOLTAN_NEXT_OBJ_FN** pair
ZOLTAN_NUM_EDGES_MULTI_F N or **ZOLTAN_NUM_EDGES_FN**
ZOLTAN_EDGE_LIST_MULTI_F N or **ZOLTAN_EDGE_LIST_FN**

Data Services and Utilities

Within Zoltan, several utilities are provided to simplify both application development and development of new algorithms in the library. They are separate from the Zoltan library so that applications can use them independently of Zoltan, if desired. They are compiled separately from Zoltan and can be archived in separate libraries. Instructions for building the utilities and applications using them are included below; individual library names are listed in the following documentation for each package.

The packages available are listed below.

- Memory Management Utilities
- Unstructured Communication Utilities
- Distributed Directory Utility

Building Utilities

The utilities provided with Zoltan have their own Makefiles and can be built separately from Zoltan. If the user builds the Zoltan library, the utility libraries are built automatically and copied to the appropriate *Zoltan/Obj_<platform>* directory, where *<platform>* is specified through the ZOLTAN_ARCH environment variable. Zoltan and the utilities share the *Utilities/Config/Config.<platform>* files specifying compilation paths for various architectures. If, however, a user wishes to use these utilities without using Zoltan, he must build the libraries separately.

The structure and use of Makefiles for the utilities are similar to Zoltan's makefiles; a top-level makefile includes rules for building each utility's library. Object files and the utility libraries are stored in a subdirectory *Obj_<platform>*, where *<platform>* is a target architecture supported with a *Utilities/Config/Config.<platform>* file. The command for compiling a particular utility follows:

gmake ZOLTAN_ARCH=*<platform>* *<library_name>*

where *<library_name>* is the name of the utility library, and *<platform>* is the target architecture (corresponding to *Utilities/Config/Config.<platform>*). The *<library_name>* for each utility is included in the following documentation for the utilities.

Building Applications

The utilities are designed so that they can easily be used separately from Zoltan in applications. To enable type-checking of arguments, the function-prototypes file for a utility should be included in all application source code files that directly access the utility. The application must also link with the appropriate utility library (and any other libraries on which the utility depends). Library and function-prototype file names for each utility are listed in the following documentation for the utilities.

[Table of Contents | Next: Memory Management Utilities | Previous: Parallel Coloring]

Memory Management Utilities

This package consists of wrappers around the standard C memory allocation and deallocation routines which add error-checking and debugging capabilities. These routines are packaged separately from Zoltan to allow their independent use in other applications. A Fortran90 interface is not yet available. C++ programmers can include the header file "zoltan_mem.h" and use the C functions. This header file, and in fact all of Zoltan's C language header files, are surrounded by an **extern "C" {}** declaration to prevent name mangling when compiled with a C++ compiler.

Source code location:	*Utilities/Memory*
Function prototypes file:	*Utilities/Memory/zoltan_mem.h* or *include/zoltan_mem.h*
Library name:	libzoltan_mem.a
Other libraries used by this library:	libmpi.a. (See note below.)

Routines:

Zoltan_Array_Alloc: Allocates arrays of dimension *n*, *n*=0,1,...,4
Zoltan_Malloc: Wrapper for system malloc.
Zoltan_Calloc: Wrapper for system calloc.
Zoltan_Realloc: Wrapper for system realloc.
Zoltan_Free: Frees memory and sets the pointer to NULL.
Zoltan_Memory_Debug: Sets the debug level used by the memory utilities; see the description below.
Zoltan_Memory_Stats: Prints memory debugging statistics, such as memory leak information.
Zoltan_Memory_Usage: Returns user-specified information about memory usage (i.e. maximum memory used, total memory currently allocated).

Use in Zoltan:

The memory management utility routines are used extensively in Zoltan and in some individual algorithms. Zoltan developers use these routines directly for most memory management, taking advantage of the error checking and debugging capabilities of the library.

Rather than call **Zoltan_Memory_Debug** directly, applications using Zoltan can set the **DEBUG_MEMORY** parameter used by this utility through calls to **Zoltan_Set_Param**.

Note on MPI usage:

MPI is used only to obtain the processor number (through a call to MPI_Comm_rank) for print statements and error messages. If an application does not link with MPI, the memory utilities should be compiled with -DZOLTAN_NO_MPI; all output will then appear to be from processor zero, even if it is actually from other processors.

---

double ***Zoltan_Array_Alloc**(char * *file*, int *line*, int *n*, int *d1*, int *d2*, ..., int *dn*, int *size*);

The **Zoltan_Array_Alloc** routine dynamically allocates an array of dimension *n*, *n* = 0, 1, ..., 4 with size (*d1* x *d2* x ... x *dn*). It is intended to be used for 2, 3 and 4 dimensional arrays; **Zoltan_Malloc** should be used for the simpler cases. The memory allocated by **Zoltan_Array_Alloc** is contiguous, and can be freed by a single call to **Zoltan_Free**.

Arguments:

file	A string containing the name of the file calling the function. The *__FILE__* macro can be passed as this argument. This argument is useful for debugging memory allocation problems.
line	The line number within *file* of the call to the function. The *__LINE__* macro can be passed as this argument. This argument is useful for debugging memory allocation problems.
n	The number of dimensions in the array to be allocated. Valid values are 0, 1, 2, 3, or 4.
d1, *d2*, ..., *dn*	The size of each dimension to be allocated. One argument is included for each dimension.
size	The size (in bytes) of the data objects to be stored in the array.

Returned Value:

double *	A pointer to the starting address of the *n*-dimensional array, or NULL if the allocation fails.

Example:

int ** *x* = (int **) **Zoltan_Array_Alloc** (*__FILE__* , *__LINE__* , 2, 5, 6, *sizeof* (int));

Allocates a two-dimensional, 5x6-element array of integers.

---

double ***Zoltan_Malloc**(int *n*, char * *file* , int *line*);

The **Zoltan_Malloc** function is a wrapper around the standard C malloc routine. It allocates a block of memory of size *n* bytes. The principle advantage of using the wrapper is that it allows memory leaks to be tracked via the DEBUG_MEMORY variable (set in **Zoltan_Memory_Debug**).

A macro **ZOLTAN_MALLOC** is defined in *zoltan_mem.h*. It takes the argument *n*, and adds the *__FILE__* and *__LINE__* macros to the argument list of the **Zoltan_Malloc** call:

#define **ZOLTAN_MALLOC**(*n*) **Zoltan_Malloc**((*n*), *__FILE__*, *__LINE__*)

Using this macro, the developer gains the file and line debugging information without having to type file and line information in each memory allocation call.

Arguments:

n	The size (in bytes) of the memory-allocation request.

file	A string containing the name of the file calling the function. The *__FILE__* macro can be passed as this argument. This argument is useful for debugging memory allocation problems.
line	The line number within *file* of the call to the function. The *__LINE__* macro can be passed as this argument. This argument is useful for debugging memory allocation problems.

Returned Value:

double *	A pointer to the starting address of memory allocated. NULL is returned if *n* = 0 or the routine is unsuccessful.

Example:

struct **Zoltan_Struct** **b* = (struct **Zoltan_Struct** *) **ZOLTAN_MALLOC**(*sizeof*(struct **Zoltan_Struct**));
Allocates memory for one **Zoltan_Struct** data structure.

---

double ***Zoltan_Calloc**(int *num*, int *size*, char * *file*, int *line*);

The **Zoltan_Calloc** function is a wrapper around the standard C calloc routine. It allocates a block of memory of size *num * size* bytes and initializes the memory to zeros. The principle advantage of using the wrapper is that it allows memory leaks to be tracked via the DEBUG_MEMORY variable (set in **Zoltan_Set_Memory_Debug**).

A macro **ZOLTAN_CALLOC** is defined in *zoltan_mem.h*. It takes the arguments *num* and *size*, and adds the *__FILE__* and *__LINE__* macros to the argument list of the **Zoltan_Calloc** call:

#define **ZOLTAN_CALLOC**(*num*, *size*) **Zoltan_Calloc**((*num*), (*size*), *__FILE__*, *__LINE__*)

Using this macro, the developer gains the file and line debugging information without having to type file and line information in each memory allocation call.

Arguments:

num	The number of elements of the following *size* to allocate.
size	The size of each element. Hence, the total allocation is *num* * *size* bytes.
file	A string containing the name of the file calling the function. The *__FILE__* macro can be passed as this argument. This argument is useful for debugging memory allocation problems.
line	The line number within *file* of the call to the function. The *__LINE__* macro can be passed as this argument. This argument is useful for debugging memory allocation problems.

Returned Value:

double *	A pointer to the starting address of memory allocated. NULL is returned if *n* = 0 or the routine is unsuccessful.

Example:

int **b* = (int *) **ZOLTAN_CALLOC**(*10*, *sizeof*(int));
Allocates memory for 10 integers and initializes the memory to zeros.

---

double ***Zoltan_Realloc**(void **ptr*, int *n*, char **file*, int *line*);

The **Zoltan_Realloc** function is a "safe" version of realloc. It changes the size of the object pointed to by *ptr* to *n* bytes. The contents of *ptr* are unchanged up to a minimum of the old and new sizes. Error tests ensuring that *n* is a positive number and that space is available to be allocated are performed.

A macro **ZOLTAN_REALLOC** is defined in *zoltan_mem.h*. It takes the arguments *ptr* and *n*, and adds the *__FILE__* and *__LINE__* macros to the argument list of the **Zoltan_Realloc** call:

```
#define   ZOLTAN_REALLOC(ptr, n)  Zoltan_Realloc((ptr), (n), __FILE__, __LINE__)
```

Using this macro, the developer gains the file and line debugging information without having to type file and line information in each memory allocation call.

Arguments:

ptr	Pointer to allocated memory to be re-sized.
n	The size (in bytes) of the memory-allocation request.
file	A string containing the name of the file calling the function. The *__FILE__* macro can be passed as this argument. This argument is useful for debugging memory allocation problems.
line	The line number within *file* of the call to the function. The *__LINE__* macro can be passed as this argument. This argument is useful for debugging memory allocation problems.

Returned Value:

double *	A pointer to the starting address of memory allocated. If the routine is unsuccessful, NULL is returned and **ptr* is unchanged.

Example:

int n = *sizeof*(struct **Zoltan_Struct**);
int **b* = (*int* *) **ZOLTAN_MALLOC** (*n*));
b = (*int* *) **ZOLTAN_REALLOC** (*b*, 2**n*);
Reallocates memory for *b* from length *n* to length 2**n*.

---

void **Zoltan_Free**(void ***ptr*, char * *file* , int *line*);

The **Zoltan_Free** function calls the system's "free" function for the memory pointed to by **ptr*. Note that the argument to this routine has an extra level of indirection when compared to the standard C "free" call. This allows the pointer being freed to be set to NULL, which can help find errors in which a pointer is used after it is deallocated. Error checking is performed to prevent attempts to free NULL pointers. When **Zoltan_Free** is used with the DEBUG_MEMORY options (set in **Zoltan_Memory_Debug**), it can help identify memory leaks.

A macro **ZOLTAN_FREE** is defined in *zoltan_mem.h*. It takes the argument *ptr*, and adds the *__FILE__* and *__LINE__* macros to the argument list of the **Zoltan_Free** call:

```
#define   ZOLTAN_FREE(ptr)   Zoltan_Free((void **)(ptr), __FILE__, __LINE__)
```

Using this macro, the developer gains the file and line debugging information without having to type file and line information in each memory allocation call.

Arguments:

ptr	Address of a pointer to the memory to be freed. Upon return, *ptr* is set to NULL.

Example:

ZOLTAN_FREE(& *x*);
Frees memory associated with the variable *x*; upon return, *x* is NULL.

Debugging Memory Errors

One important reason to use the memory-management utilities' wrappers around the system memory routines is to facilitate debugging of memory problems. Various amounts of information can about memory allocation and deallocation are stored, depending on the debug level set through a call to **Zoltan_Memory_Debug**. This information is printed either when an error or warning occurs, or when **Zoltan_Memory_Stats** is called. We have found values of one and two to be very helpful in our development efforts. The routine **Zoltan_Memory_Usage** can be called to return user-specified information about memory utilization to the user's program.

---

void **Zoltan_Memory_Debug**(int *new_level*);

The **Zoltan_Memory_Debug** function sets the level of memory debugging to be used.

Arguments:

new_level — Integer indicating the amount of debugging to use. Valid options include:

0 -- No debugging.
1 -- The number of calls to **Zoltan_Malloc** and **Zoltan_Free** are tallied, and can be printed by a call to **Zoltan_Memory_Stats**.
2 -- A list of all calls to **Zoltan_Malloc** which have not yet been freed is kept. This list is printed by **Zoltan_Memory_Stats** (useful for detecting memory leaks). Any calls to **Zoltan_Free** with addresses not in this list trigger warning messages. (Note that allocations that occurred prior to setting the debug level to 2 will not be in this list and thus can generate spurious warnings.)
3 -- Information about each allocation is printed as it happens.

Default:

Memory debug level is 1.

---

void **Zoltan_Memory_Stats**();

The **Zoltan_Memory_Stats** function prints information about memory allocation and deallocation. The amount of information printed is determined by the debug level set through a call to **Zoltan_Memory_Debug**.

Arguments:

None.

---

int **Zoltan_Memory_Usage**(int *type*);

The **Zoltan_Memory_Usage** function returns information about memory utilization. The memory debug level (set through a call to **Zoltan_Set_Memory_Debug**) must be at least 2 for this function to return non-zero values.

Arguments:

type — Integer to request type of information required. These integers are defined in *zoltan_mem.h*. Valid options include:

ZOLTAN_MEM_STAT_TOTAL -- The function will return the current total memory allocated via Zoltan's memory allocation routines.
ZOLTAN_MEM_STAT_MAXIMUM -- The function will return the maximum total memory allocated via Zoltan's memory allocation routines up to this point.

Default:

type = *ZOLTAN_MEM_STAT_MAXIMUM*

Returned Value:

int — The number in bytes of the specific requested memory statistic.

Example:

total = **Zoltan_Memory_Usage** (*ZOLTAN_MEM_STAT_TOTAL*);

[Table of Contents | Next: Unstructured Communication Utilities | Previous: Utilities]

Unstructured Communication Utilities

The unstructured communication package provides a simple interface for doing complicated patterns of point-to-point communication, such as those associated with data remapping. This package consists of a few simple functions which create or modify communication *plans*, perform communication, and destroy communication plans upon completion. The package is descended from software first developed by Steve Plimpton and Bruce Hendrickson, and has proved useful in a variety of different applications. For this reason, it is maintained as a separate library and can be used independently from Zoltan.

In a prototypical usage of the unstructured communication package each processor has some objects to send to other processors, but no processor knows what messages it will receive. A call to **Zoltan_Comm_Create** produces a data structure called a communication *plan* which encapsulates the basic information about the communication operation. The plan does not know anything about the types of objects being transferred, only the number of them. So the same plan can be used repeatedly to transfer different types of data as long as the number of objects in the transfers remains the same. The actual size of objects isn't specified until the call to **Zoltan_Comm_Do** which performs the data transfer.

The plan which is produced by **Zoltan_Comm_Create** assumes that all the objects are of the same size. If this is not true, then a call to **Zoltan_Comm_Resize** can specify the actual size of each object, and the plan is augmented appropriately. **Zoltan_Comm_Resize** can be invoked repeatedly on the same plan to specify varying sizes for different data transfer operations.

Although a friendlier interface may be added in the future, for now the data to be sent must be passed to **Zoltan_Comm_Do** as a packed buffer in which the objects are stored consecutively. This probably requires the application to pull the data out of native data structures and place in into the buffer. The destination of each object is specified by the *proclist* argument to **Zoltan_Comm_Create**. Some flexibility is supported by allowing *proclist* to contain negative values, indicating that the corresponding objects are not to be sent. The communication operations allow for any object to be sent to any destination processor. However, if the objects are grouped in such a way that all those being sent to a particular processor are consecutive, the time and memory of an additional copy is avoided.

Function **Zoltan_Comm_Do_Reverse** reverses the communication plan to send back messages to the originators.

To allow overlap between communication and processing, POST and WAIT variants of **Zoltan_Comm_Do** and **Zoltan_Comm_Do_Reverse** are provided. Communication is initiated by the POST function (**Zoltan_Comm_Do_Post** or **Zoltan_Comm_Do_Reverse_Post**); incoming messages are posted and outgoing messages are sent. Then the user can continue processing. After the processing is complete, the corresponding WAIT function (**Zoltan_Comm_Do_Wait** or **Zoltan_Comm_Do_Reverse_Wait**) is called to wait for all incoming messages to be received. For convenience, these functions use the same calling arguments as **Zoltan_Comm_Do** and **Zoltan_Comm_Do_Reverse**.

All the functions in the unstructured communication library return integer error codes identical to those used by Zoltan.

The C++ interface to the unstructured communication utility is found in the *zoltan_comm_cpp.h* header file which defines the **Zoltan_Comm** class.

A Fortran90 interface is not yet available.

Source code location:	*Utilities/Communication*
C Function prototypes file:	*Utilities/Communication/zoltan_comm.h*
C++ class definition:	*Utilities/Communication/zoltan_comm_cpp.h*
Library name:	libzoltan_comm.a
Other libraries used by this library:	libmpi.a, libzoltan_mem.a.

High Level Routines:

Zoltan_Comm_Create: computes a communication plan for sending objects to destination processors.
Zoltan_Comm_Do: uses a communication plan to send data objects to destination processors. The POST and WAIT variants are
Zoltan_Comm_Do_Post and
Zoltan_Comm_Do_Wait.
Zoltan_Comm_Do_Reverse: performs the reverse (opposite) communication of **Zoltan_Comm_Do**. The POST and WAIT variants are
Zoltan_Comm_Do_Reverse_Post and
Zoltan_Comm_Do_Reverse_Wait.
Zoltan_Comm_Resize: augments the plan to allow objects to be of variable sizes.
Zoltan_Comm_Copy: create a new communication plan and copy an existing one to it.
Zoltan_Comm_Copy_To: copy one existing communication plan to another.
Zoltan_Comm_Destroy: free memory associated with a communication plan.

Low Level Routines:

Zoltan_Comm_Exchange_Sizes: updates the sizes of the messages each processor will receive.
Zoltan_Comm_Invert_Map: given a set of messages each processor wants to send, determines the set of messages each processor needs to receive.
Zoltan_Comm_Sort_Ints: sorts an array of integer values.
Zoltan_Comm_Info: returns information about a communication plan.
Zoltan_Comm_Invert_Plan: given a communication plan, converts the plan into a plan for the reverse communication.

Use in Zoltan:

The Zoltan library uses the unstructured communication package in its migration tools and in some of the load-balancing algorithms. For example, in **Zoltan_Migrate**, **Zoltan_Comm_Create** is used to develop a communication map for sending objects to be exported to their new destination processors. The sizes of the exported objects are obtained and the communication map is augmented with a call to **Zoltan_Comm_Resize**. The data for the objects is packed into a communication buffer and sent to the other processors through a call to **Zoltan_Comm_Do**. After the received objects are unpacked, the communication plan is no longer needed, and it is deallocated by a call to **Zoltan_Comm_Destroy**. Zoltan developers use the package whenever possible, as improvements made to the package (for example, support for heterogeneous architectures) automatically propagate to the algorithms.

C:
int **Zoltan_Comm_Create**(struct Zoltan_Comm_Obj ***plan*, int *nsend*, int **proclist*, MPI_Comm *comm*, int *tag*, int **nreturn*);
C++:
Zoltan_Comm(const int & *nsend*, int **proclist*, const MPI_Comm & *comm*, const int & *tag*, int **nreturn*);
or
Zoltan_Comm();
Zoltan_Comm::Create(const int & *nsend*, int **proclist*, const MPI_Comm & *comm*, const int & *tag*, int **nreturn*);

The **Zoltan_Comm_Create** function sets up the communication plan in the unstructured communication package. Its input is a count of objects to be sent to other processors, a list of the processors to which the objects should be sent (repetitions are allowed), and an MPI communicator and tag. It allocates and builds a communication plan that describes to which processors data will be sent and from which processors data will be received. It also computes the amount of data to be sent to and received from each processor. It returns the number of objects to be received by the processor and a pointer to the communication plan it created. The communication plan is then used by calls to **Zoltan_Comm_Do** to perform the actual communication.

Arguments:

plan	A pointer to the communication plan created by **Zoltan_Comm_Create**.
nsend	The number of objects to be sent to other processors.
proclist	An array of size *nsend* of destination processor numbers for each of the objects to be sent.
comm	The MPI communicator for the unstructured communication.
tag	A tag for MPI communication.
nreturn	Upon return, the number of objects to be received by the processor.

Returned Value:

int	Error code.

In the C++ interface to the communication utility, the communication plan is represented by a **Zoltan_Comm** object. It is created when the **Zoltan_Comm** constructor executes. There are two constructors. The first one listed above uses parameters to initialize the plan. The second constructor does not, but the plan can subsequently be initialized with a call to **Zoltan_Comm::Create()**.

C:
int **Zoltan_Comm_Do**(struct Zoltan_Comm_Obj **plan*, int *tag*, char **send_data*, int *nbytes*, char **recvbuf*);
int **Zoltan_Comm_Do_Post**(struct Zoltan_Comm_Obj **plan*, int *tag*, char **send_data*, int *nbytes*, char **recvbuf*);
int **Zoltan_Comm_Do_Wait**(struct Zoltan_Comm_Obj **plan*, int *tag*, char **send_data*, int *nbytes*, char **recvbuf*);
C++:
int **Zoltan_Comm::Do**(const int & *tag*, char **send_data*, const int & *nbytes*, char **recvbuf*);
int **Zoltan_Comm::Do_Post**(const int & *tag*, char **send_data*, const int & *nbytes*, char **recvbuf*);
int **Zoltan_Comm::Do_Wait**(const int & *tag*, char **send_data*, const int & *nbytes*, char **recvbuf*);

The **Zoltan_Comm_Do** function performs the communication described in a communication plan built by **Zoltan_Comm_Create**. Using the plan, it takes a buffer of object data to be sent and the size (in bytes) of each object's data in that buffer and sends the data to other processors. **Zoltan_Comm_Do** also receives object data from other processors and stores it in a receive buffer. The receive buffer must be allocated by the code calling **Zoltan_Comm_Do** using the number of received objects returned by **Zoltan_Comm_Create** or **Zoltan_Comm_Resize**. If the objects have variable sizes, then **Zoltan_Comm_Resize** must be called before **Zoltan_Comm_Do**.

Arguments:	
plan	A pointer to a communication plan built by **Zoltan_Comm_Create**.
tag	An MPI message tag.
send_data	A buffer filled with object data to be sent to other processors.
nbytes	The size (in bytes) of the data for one object, or the scale factor if the objects have variable sizes. (See **Zoltan_Comm_Resize** for more details.)
recvbuf	Upon return, a buffer filled with object data received from other processors.
Returned Value:	
int	Error code.

C:
int **Zoltan_Comm_Do_Reverse**(struct Zoltan_Comm_Obj **plan*, int *tag*, char **send_data*, int *nbytes*, int **sizes*, char **recvbuf*);
int **Zoltan_Comm_Do_Reverse_Post**(struct Zoltan_Comm_Obj **plan*, int *tag*, char **send_data*, int *nbytes*, int **sizes*, char **recvbuf*);
int **Zoltan_Comm_Do_Reverse_Wait**(struct Zoltan_Comm_Obj **plan*, int *tag*, char **send_data*, int *nbytes*, int **sizes*, char **recvbuf*);
C++:

int **Zoltan_Comm::Do_Reverse**(const int & *tag*, char **send_data*, const int & *nbytes*, int **sizes*, char **recvbuf*);
int **Zoltan_Comm::Do_Reverse_Post**(const int & *tag*, char **send_data*, const int & *nbytes*, int **sizes*, char **recvbuf*);
int **Zoltan_Comm::Do_Reverse_Wait**(const int & *tag*, char **send_data*, const int & *nbytes*, int **sizes*, char **recvbuf*);

The **Zoltan_Comm_Do_Reverse** function performs communication based on a communication plan built by **Zoltan_Comm_Create**. But unlike **Zoltan_Comm_Do**, this routine performs the reverse of the communication pattern. Specifically, all sends in the plan are treated as receives and vice versa. **Zoltan_Comm_Do_Reverse** is particularly well suited to return updated data objects to their originating processors when the objects were initially transferred via **Zoltan_Comm_Do**.

Arguments:	
plan	A pointer to a communication plan built by **Zoltan_Comm_Create**.
tag	An MPI message tag to be used by this routine.
send_data	A buffer filled with object data to be sent to other processors.
nbytes	The size (in bytes) of the data associated with an object, or the scale factor if the objects have variable sizes.
sizes	If not NULL, this input array specifies the size of all the data objects being transferred. This argument is passed directly to **Zoltan_Comm_Resize**. This array has length equal to the *nsend* value passed to **Zoltan_Comm_Create**. But note that for **Zoltan_Comm_Do_Reverse** this array describes the sizes of the values being received, not sent.
recvbuf	Upon return, a buffer filled with object data received from other processors.
Returned Value:	
int	Error code.

C:
int **Zoltan_Comm_Resize**(struct Zoltan_Comm_Obj **plan*, int **sizes*, int *tag* , int **total_recv_size*);
C++:
int **Zoltan_Comm::Resize**(int **sizes*, const int & *tag* , int **total_recv_size*);

If the objects being communicated are of variable sizes, then the plan produced by **Zoltan_Comm_Create** is incomplete. This routine allows the plan to be augmented to allow for variable sizes. **Zoltan_Comm_Resize** can be invoked repeatedly on the same plan to specify different object sizes associated with different data transfers.

Arguments:	
plan	A communication plan built by **Zoltan_Comm_Create**.

sizes	An input array of length equal to the *nsend* argument in the call to **Zoltan_Comm_Create** which generated the *plan*. Each entry in the array is the size of the corresponding object to be sent. If *sizes* is *NULL* (on all processors), the objects are considered to be the same size. Note that the true size of a message will be scaled by the *nbytes* argument to **Zoltan_Comm_Do**.
tag	A message tag to be used for communication within this routine, based upon the communicator in *plan*.
total_recv_size	Sum of the sizes of the incoming messages. To get the actual size (in bytes), you need to scale by the *nbytes* argument to **Zoltan_Comm_Do**.

Returned Value:

int	Error code.

C: struct Zoltan_Comm_Obj ***Zoltan_Comm_Copy**(struct Zoltan_Comm_Obj **plan*);
C++: **Zoltan_Comm(const Zoltan_Comm &plan);**

Zoltan_Comm_Copy creates a new Zoltan_Comm_Obj structure and copies the existing *plan* to it. The corresponding C++ method is the **Zoltan_Comm** copy constructor.

Arguments:

plan	A pointer to the communication plan to be copied to the new Zoltan_Comm_Obj structure.

Returned Value:

struct Zoltan_Comm_Obj *	the newly created plan, or NULL on error.

C: int **Zoltan_Comm_Copy_To**(struct Zoltan_Comm_Obj ***to*, struct Zoltan_Comm_Obj **from*);
C++: Zoltan_Comm & **operator=** (const Zoltan_Comm &plan);

Zoltan_Comm_Copy_To copies one existing communication plan to another. The corresponding C++ method is the **Zoltan_Comm** copy operator.

Arguments:

to	A pointer to a pointer to the communication plan that will be copied to. We destroy the plan first, and set the pointer to the plan to NULL, before proceeding with the copy.
from	A pointer the communication plan that we will make a copy of.

Returned Value:

int	Error code

C: int **Zoltan_Comm_Destroy**(struct Zoltan_Comm_Obj ***plan*);
C++: **~Zoltan_Comm();**

The **Zoltan_Comm_Destroy** function frees all memory associated with a communication plan created by **Zoltan_Comm_Create**. The C++ **Zoltan_Comm** object does not have an explicit **Destroy** method. It is deallocated when it's destructor is called.

Arguments:

plan	A pointer to a communication plan built by **Zoltan_Comm_Create**. Upon return, *plan* is set to NULL.

Returned Value:

int	Error code.

C:
int **Zoltan_Comm_Exchange_Sizes**(int **sizes_to*, int **procs_to*, int *nsends*, int *self_msg*, int **sizes_from*, int **procs_from*, int *nrecvs*, int **total_recv_size*, int *my_proc*, int *tag*, MPI_Comm *comm*);
C++:
static int **Zoltan_Comm::Exchange_Sizes**(int **sizes_to*, int **procs_to*, const int & *nsends*, const int & *self_msg*, int **sizes_from*, int **procs_from*, const int & *nrecvs*, int **total_recv_size*, const int & *my_proc*, const int & *tag*, const MPI_Comm & *comm*);

This routine is used by **Zoltan_Comm_Resize** to update the sizes of the messages each processor is expecting to receive. The processors already know who will send them messages, but if variable sized objects are being communicated, then the sizes of the messages are recomputed and exchanged via this routine.

Arguments:

sizes_to	Input array with the size of each message to be sent. Note that the actual number of bytes in the message is the product of this value and the *nbytes* argument to **Zoltan_Comm_Do**.
procs_to	Input array with the destination processor for each of the messages to be sent.
nsends	Input argument with the number of messages to be sent. (Length of the *procs_to* array.)
self_msg	Input argument indicating whether a processor has data for itself (=1) or not (=0) within the *procs_to* and *lengths_to* arrays.
sizes_from	Returned array with the size of each message that will be received. Note that the actual number of bytes in the message is the product of this value and the *nbytes* argument to **Zoltan_Comm_Do**.
procs_from	Returned array of processors from which data will be received.
nrecvs	Returned value with number of messages to be received. (length of *procs_from* array.)
total_recv_size	The total size of all the messages to be received. As above, the actual number of bytes will be scaled by the *nbytes* argument to **Zoltan_Comm_Do**.
my_proc	The processor's ID in the *comm* communicator.
tag	A message tag which can be used by this routine.
comm	MPI Communicator for the processor numbering in the *procs* arrays.

Returned Value:	
int	Error code.

C:
int **Zoltan_Comm_Invert_Map**(int **lengths_to*, int * *procs_to*, int *nsends*, int *self_msg*, int ** *lengths_from*, int ** *procs_from*, int * *nrecvs*, int *my_proc*, int *nprocs*, int *out_of_mem*, int *tag*, MPI_Comm *comm*);
C++:
static int **Zoltan_Comm::Invert_Map**(int **lengths_to*, int * *procs_to*, const int & *nsends*, const int & *self_msg*, int * & *lengths_from*, int * & *procs_from*, int & *nrecvs*, const int & *my_proc*, const int & *nprocs*, const int & *out_of_mem*, const int & *tag*, const MPI_Comm & *comm*);

The **Zoltan_Comm_Invert_Map** function is a low level communication routine. It is useful when a processor knows to whom it needs to send data, but not from whom it needs to receive data. Each processor provides to this routine a set of lengths and destinations for the messages it wants to send. The routine then returns the set of lengths and origins for the messages a processor will receive. Note that by inverting the interpretation of *to* and *from* in these arguments, the routine can be used to do the opposite: knowing how much data to receive and from which processors, it can compute how much data to send and to which processors.

Arguments:	
lengths_to	Input array with the number of values in each of the messages to be sent. Note that the actual size of each value is not specified until the **Zoltan_Comm_Do** routine is invoked.
procs_to	Input array with the destination processor for each of the messages to be sent.
nsends	Input argument with the number of messages to be sent. (Length of the *lengths_to* and *procs_to* arrays.)
self_msg	Input argument indicating whether a processor has data for itself (=1) or not (=0) within the *procs_to* and *lengths_to* arrays.
lengths_from	Returned array with lengths of messages to be received.
procs_from	Returned array of processors from which data will be received.
nrecvs	Returned value with number of messages to be received (lengths of *lengths_from* and *procs_from* arrays).
my_proc	The processor's ID in the *comm* communicator.
nprocs	Number of processors in the *comm* communicator.
out_of_mem	Since it has a barrier operation, this routine is a convenient time to tell all the processors that one of them is out of memory. This input argument is 0 if the processor is OK, and 1 if the processor has failed in a malloc call. All the processors will return with a code of **COMM_MEMERR** if any of them is out of memory.
tag	A message tag which can be used by this routine.
comm	MPI Communicator for the processor numbering in the *procs* arrays.
Returned Value:	
int	Error code.

---

int **Zoltan_Comm_Sort_Ints**(int **vals_sort*, int **vals_other*, int *nvals*);

As its name suggests, the **Zoltan_Comm_Sort_Ints** function sorts a set of integers via the quicksort algorithm. The integers are reordered from lowest to highest, and a second array of integers is reordered in the same fashion. This second array can be used to return the permutation associated with the sort operation. There is no C++ interface to this function. You can use the C function instead.

Arguments:	
vals_sort	The array of integers to be sorted. This array is permuted into sorted order.
vals_other	Another array of integers which is permuted identically to *vals_sort*.
nvals	The number of values in the two integer arrays.
Returned Value:	
int	Error code.

C:
int **Zoltan_Comm_Info**(struct Zoltan_Comm_Obj **plan*, int **nsends*, int **send_procs*, int **send_lengths*, int **send_nvals*, int **send_max_size*, int **send_list*, int **nrecvs*, int **recv_procs*, int **recv_lengths*, int **recv_nvals*, int **recv_total_size*, int **recv_list*, int **self_msg*);
C++:
int **Zoltan_Comm::Info**(int **nsends*, int **send_procs*, int **send_lengths*, int **send_nvals*, int **send_max_size*, int **send_list*, int **nrecvs*, int **recv_procs*, int **recv_lengths*, int **recv_nvals*, int **recv_total_size*, int **recv_list*, int **self_msg*) const;

Zoltan_Comm_Info returns information about a communication plan. All arguments, except the *plan* itself, may be NULL; values are returned only for non-NULL arguments.

Arguments:	
plan	Communication data structure created by **Zoltan_Comm_Create**.
nsends	Upon return, the number of processors to which messages are sent; does not include self-messages.
send_procs	Upon return, a list of processors to which messages are sent; self-messages are included.
send_lengths	Upon return, the number of values to be sent to each processor in *send_procs*.
send_nvals	Upon return, the total number of values to send.
send_max_size	Upon return, the maximum size of a message to be sent; does not include self-messages.
send_list	Upon return, the processor assignment of each value to be sent.
nrecvs	Upon return, the number of processors from which to receive messages; does not include self-messages.
recv_procs	Upon return, a list of processors from which messages are received; includes self-messages.
recv_lengths	Upon return, the number of values to be received from each processor in *recv_procs*.

recv_nvals	Upon return, the total number of values to receive.
recv_total_size	Upon return, the total size of items to be received.
recv_list	Upon return, the processor assignments of each value to be received.
self_msg	Upon return, the number of self-messages.
Returned Value:	
int	Error code.

C: int **Zoltan_Comm_Invert_Plan**(struct Zoltan_Comm_Obj ***plan*);
C++: int **Zoltan_Comm::Invert_Plan**();

Given a communication plan, **Zoltan_Comm_Invert_Plan** alters the plan to make it the plan for the reverse communication. Information in the input plan is replaced by information for the reverse-communication plan. All receives in the reverse-communication plan are blocked; thus, using the inverted plan does not produce the same results as **Zoltan_Comm_Do_Reverse**. If an error occurs within **Zoltan_Comm_Invert_Plan**, the original plan is returned unaltered.

Arguments:	
plan	Communication data structure created by **Zoltan_Comm_Create**; the contents of this plan are irretrievably modified by **Zoltan_Comm_Invert_Plan**.
Returned Value:	
int	Error code.

[Table of Contents | Next: Distributed Directory Utility | Previous: Memory Management Utilities]

Distributed Directory Utility

The owner (i.e. the processor number) of any computational object is subject to change during load balancing. An application may use this directory utility to manage its objects' locations. A distributed directory balances the load (in terms of memory and processing time) and avoids the bottle neck of a centralized directory design.

This distributed directory module may be used alone or in conjunction with Zoltan's load balancing capability and memory and communication services. The user should note that external names (subroutines, etc.) which prefaced by Zoltan_DD_ are reserved when using this module.

The user initially creates an empty distributed directory using Zoltan_DD_Create. Then global ID (GID) information is added to the directory using Zoltan_DD_Update. The directory maintains the GID's basic information: local ID (optional), partition (optional), arbitrary user data (optional), and the current data owner. Zoltan_DD_Update is also called after data migration or refinements. Zoltan_DD_Find returns the directory information for a list of GIDs. A selected list of GIDs may be removed from the directory by Zoltan_DD_Remove. When the user has finished using the directory, its memory is returned to the system by Zoltan_DD_Destroy.

An object is known by its GID. Hashing provides very fast lookup for the information associated with a GID in a two step process. The first hash of the GID yields the processor number owning the directory entry for that GID. The directory entry owner remains constant even if the object (GID) migrates in time. Second, a different hash algorithm of the GID looks up the associated information in directory processor's hash table. The user may optionally register their own (first) hash function to take advantage of their knowledge of their GID naming scheme and the GID's neighboring processors. See the documentation for Zoltan_DD_Set_Hash_Fn for more information. If no user hash function is registered, Zoltan's **Zoltan_Hash** will be used. This module's design was strongly influenced by the paper "Communication Support for Adaptive Computation" by Pinar and Hendrickson.

Some users number their GIDs by giving the first "n" GIDs to processor 0, the next "n" GIDs to processor 1, and so forth. The function Zoltan_DD_Set_Neighbor_Hash_Fn1 will provide efficient directory communication when these GIDs stay close to their origin. The function Zoltan_DD_Set_Neighbor_Hash_Fn2 allows the specification of ranges of GIDs to each processor for more flexibility. The source code for DD_Set_Neighbor_Hash_Fn1 and DD_Set_Neighbor_Hash_Fn2 provide examples of how a user can create their own "hash" functions taking advantage of their own GID naming convention.

The routine Zoltan_DD_Print will print the contents of the directory. The companion routine Zoltan_DD_Stats prints out a summary of the hash table size, number of linked lists, and the length of the longest linked list. This may be useful when the user creates their own hash functions.

The C++ interface to this utility is defined in the header file *zoltan_dd_cpp.h* as the class **Zoltan_DD**. A single **Zoltan_DD** object represents a distributed directory.

A Fortran90 interface is not yet available.

Source code location:	*Utilities/DDirectory*
C Function prototypes file:	*Utilities/DDirectory/zoltan_dd.h*
C++ class definition:	*Utilities/DDirectory/zoltan_dd_cpp.h*
Library name:	libzoltan_dd.a
Other libraries used by this library:	libmpi.a, libzoltan_mem.a, libzoltan_comm.a

Routines:

Zoltan_DD_Create: Allocates memory and initializes the directory.
Zoltan_DD_Copy: Allocates a new directory structure and copies an existing one to it.
Zoltan_DD_Copy_To: Copies one directory structure to another.
Zoltan_DD_Destroy: Terminate the directory and frees its memory.
Zoltan_DD_Update: Adds or updates GIDs' directory information.
Zoltan_DD_Find: Returns GIDs' information (owner, local ID, etc.)
Zoltan_DD_Remove: Eliminates selected GIDs from the directory.
Zoltan_DD_Stats: Provides statistics about hash table & linked lists.
Zoltan_DD_Print: Displays the contents (GIDs, etc) of each directory.
Zoltan_DD_Set_Hash_Fn: Registers a user's optional hash function.
Zoltan_DD_Set_Neighbor_Hash_Fn1: Hash function with constant number of GIDs per processor.
Zoltan_DD_Set_Neighbor_Hash_Fn2: Hash function with variable number of GID's per processor.

Data Stuctures:

struct Zoltan_DD_Struct: State & storage used by all DD routines. Users should not modify any internal values in this structure. Users should only pass the address of this structure to the other routines in this package.

C:
int **Zoltan_DD_Create** (struct Zoltan_DD_Struct ***dd*, MPI_Comm *comm*, int *num_gid_entries*, int *num_lid_entries*, int *user_length*, int *table_length*, int *debug_level*);
C++:
Zoltan_DD(const MPI_Comm & *comm*, const int & *num_gid_entries*, const int & *num_lid_entries*, const int & *user_length*, const int & *table_length*, const int & *debug_level*);
or
Zoltan_DD();
Zoltan_DD::Create(const MPI_Comm & *comm*, const int & *num_gid_entries*, const int & *num_lid_entries*, const int & *user_length*, const int & *table_length*, const int & *debug_level*);

Zoltan_DD_Create allocates and initializes memory for the Zoltan_DD_Struct structure. It must be called before any other distributed directory routines. MPI must be initialized prior to calling this routine.

The Zoltan_DD_Struct must be passed to all other distributed directory routines. The MPI Comm argument designates the processors used for the distributed directory. The MPI Comm argument is duplicated and stored for later use.

The user can set the debug level argument in the **Zoltan_DD_Create** to determine the module's response to multiple updates for any GID within one update cycle. If the argument is set to 0, all multiple updates are ignored (but the last determines the directory information.) If the argument is set to 1, an error is returned if the multiple updates represent different owners for the same GID. If the debug level is 2, an error return and an error message are generated if multiple updates represent different owners for the same GID. If the level is 3, an error return and an error message are generated for a multiple update even if the updates represent the same owner for a GID.

Arguments:	
dd	Structure maintains directory state and hash table.
comm	MPI comm duplicated and stored specifying directory processors.
num_gid_entries	Length of GID.
num_lid_entries	Length of local ID or zero to ignore local IDs.
user_length	Length of user defined data field (optional, may be zero).
table_length	Length of hash table (zero forces default value).
debug_level	Legal values range in [0,3]. Sets response to various error conditions where 3 is the most verbose.
Returned Value:	
int	Error code.

In the C++ interface, the distributed directory is represented by a **Zoltan_DD** object. It is created when the **Zoltan_DD** constructor executes. There are two constructors. The first one listed above uses parameters to initialize the distributed directory. The second constructor does not, but it can subsequently be initialized with a call to **Zoltan_DD::Create()**.

C: struct Zoltan_DD_Struct *__Zoltan_DD_Copy__ (struct Zoltan_DD_Struct **from*);
C++: **Zoltan_DD**(const Zoltan_DD &dd);

This routine creates a new distributed directory structure and copies an existing one to it. The corresponding routine in the C++ library is the Zoltan_DD copy constructor.

Arguments:	
from	The existing directory structure which will be copied to the new one.
Returned Value:	
struct Zoltan_DD_Struct *	The newly created directory structure.

C: int **Zoltan_DD_Copy_To** (struct Zoltan_DD_Struct ***to*, struct Zoltan_DD_Struct **from*);
C++: Zoltan_DD & **operator=**(const Zoltan_DD &dd);

This routine copies one distributed directory structure to another. The corresponding method in the C++ library is the Zoltan_DD class copy operator.

Arguments:

to	A pointer to a pointer to the target structure. The structure will be destroyed and the pointer set to NULL before proceeding with the copy.
from	A pointer to the source structure. The contents of this structure will be copied to the target structure.

Returned Value:

int	Error code.

C: void **Zoltan_DD_Destroy** (struct Zoltan_DD_Struct ***dd*);
C++: **~Zoltan_DD();**

This routine frees all memory allocated for the distributed directory. No calls to any distributed directory functions using this Zoltan_DD_Struct are permitted after calling this routine. MPI is necessary for this routine only to free the previously saved MPI comm.

Arguments:

dd	Directory structure to be deallocated.

Returned Value:

void	NONE

There is no explicit **Destroy** method in the C++ **Zoltan_DD** class. The object is deallocated when it's destructor is called.

C:
int **Zoltan_DD_Update** (struct Zoltan_DD_Struct **dd*, ZOLTAN_ID_PTR *gid*, ZOLTAN_ID_PTR *lid*, ZOLTAN_ID_PTR *user*, int **partition*, int *count*);
C++:
int **Zoltan_DD::Update**(ZOLTAN_ID_PTR *gid*, ZOLTAN_ID_PTR *lid*, ZOLTAN_ID_PTR *user*, int **partition*, const int & *count*);

Zoltan_DD_Update takes a list of GIDs and corresponding lists of optional local IDs, optional user data, and optional partitions. This routine updates the information for existing directory entries or creates a new entry (filled with given data) if a GID is not found. NULL lists should be passed for optional arguments not desired. This function should be called initially and whenever objects are migrated to keep the distributed directory current.

The user can set the debug level argument in **Zoltan_DD_Create** to determine the module's response to multiple updates for any GID within one update cycle.

Arguments:

dd	Distributed directory structure state information.
gid	List of GIDs to update (in).
lid	List of corresponding local IDs (optional) (in).
user	List of corresponding user data (optional) (in).

partition	List of corresponding partitions (optional) (in).
count	Number of GIDs in update list.
Returned Value:	
int	Error code.

C:
int **Zoltan_DD_Find** (Zoltan_DD_DDirectory **dd*, ZOLTAN_ID_PTR *gid*, ZOLTAN_ID_PTR *lid*, ZOLTAN_ID_PTR *data*, int **partition*, int *count*, int **owner*);
C++:
int **Zoltan_DD::Find**(ZOLTAN_ID_PTR *gid*, ZOLTAN_ID_PTR *lid*, ZOLTAN_ID_PTR *data*, int **partition*, const int & *count*, int **owner*) const;

Given a list of GIDs, **Zoltan_DD_Find** returns corresponding lists of the GIDs' owners, local IDs, partitions, and optional user data. NULL lists must be provided for optional information not being used.

Arguments:	
dd	Distributed directory structure state information.
gid	List of GIDs whose information is requested.
lid	Corresponding list of local IDs (optional) (out).
data	Corresponding list of user data (optional) (out).
partition	Corresponding list of partitions (optional) (out).
count	Count of GIDs in above list.
owner	Corresponding list of data owners (out).
Returned Value:	
int	Error code.

C:
int **Zoltan_DD_Remove** (struct Zoltan_DD_Struct **dd*, ZOLTAN_ID_PTR *gid*, int *count*);
C++:
int **Zoltan_DD::Remove**(ZOLTAN_ID_PTR *gid*, const int & *count*);

Zoltan_DD_Remove takes a list of GIDs and removes all of their information from the distributed directory.

Arguments:	
dd	Distributed directory structure state information.
gid	List of GIDs to eliminate from the directory.
count	Number of GIDs to be removed.
Returned Value:	
int	Error code.

C:
void **Zoltan_DD_Set_Hash_Fn** (struct Zoltan_DD_Struct *dd*, unsigned int (*hash*)(ZOLTAN_ID_PTR, int, unsigned int));
C++:
void **Zoltan_DD::Set_Hash_Fn**(unsigned int (*hash*) (ZOLTAN_ID_PTR, int, unsigned int));

Enables the user to register a new hash function for the distributed directory. (If this routine is not called, the default hash function **Zoltan_Hash** will be used automatically.) This hash function determines which processor maintains the distributed directory entry for a given GID. Inexperienced users do not need this routine.

Experienced users may elect to create their own hash function based on their knowledge of their GID naming scheme. The user's hash function must have calling arguments compatible with **Zoltan_Hash**. Consider that a user has defined a hash function, myhash, as

```
unsigned int myhash(ZOLTAN_ID_PTR gid, int length, unsigned int naverage)
    {
    return *gid / naverage ; /* GID length assumed to be 1 ; naverage = total_GIDS/nproc */
    }
```

Then the call to register this hash function is:

```
Zoltan_DD_Set_Hash (myhash) ;
```

NOTE: This hash function might group the gid's directory information near the gid's owning processor's neighborhood, for an appropriate naming scheme.

Arguments:	
dd	Distributed directory structure state information.
hash	Name of user's hash function.
Returned Value:	
void	NONE

C:
void **Zoltan_DD_Stats** (struct Zoltan_DD_Struct *dd*);
C++:
void **Zoltan_DD::Stats**() const;

This routine prints out summary information about the local distributed directory. It includes the hash table length, number of GIDs stored in the local directory, the number of linked lists, and the length of the longest linked list. The debug level (set by an argument to **Zoltan_DD_Create** controls this routine's verbosity.

Arguments:	
dd	Distributed directory structure for state information
Returned Value:	
void	NONE

---

int **Zoltan_DD_Set_Neighbor_Hash_Fn1** (struct Zoltan_DD_Struct **dd*, int *size*);

This routine associates the first size GIDs to proc 0, the next size to proc 1, etc. It assumes the GIDs are consecutive numbers. It assumes that GIDs primarily stay near their original owner. The GID length is assumed to be 1. GIDs outside of the range are evenly distributed among the processors via modulo(number of processors). This is a model for the user to develop their own similar routine.

Arguments:	
dd	Distributed directory structure state information.
size	Number of consecutive GIDs associated with a processor.
Returned Value:	
int	Error code.

---

int **Zoltan_DD_Set_Neighbor_Hash_Fn2** (struct Zoltan_DD_Struct *dd, int **proc*, int **low*, int **high*, int *n*);

This routine allows the user to specify a beginning and ending GID "numbers" per directory processor. It assumes that GIDs primarily stay near their original owner. It requires that the numbers of high, low, & proc entries are all n. It assumes the GID length is 1. It is a model for the user to develop their own similar routine. Users should note the registration of a cleanup routine to free local static memory when the distributed directory is destroyed. GIDs outside the range specified by high and low lists are evenly distributed among the processors via modulo (number of processors).

Arguments:	
dd	Distributed directory structure state information.
proc	List of processor ids labeling for corresponding high, low value.
low	List of low GID limits corresponding to proc list.
high	List of high GID limits corresponding to proc list.
n	Number of elements in the above lists. Should be number of processors!
Returned Value:	
int	Error code.

C: int **Zoltan_DD_Print** (struct Zoltan_DD_Struct **dd*);
C++: int **Zoltan_DD::Print** () const;

This utility displays (to stdout) the entire contents of the distributed directory at one line per GID.

Arguments:	
dd	Distributed directory structure state information.
Returned Value:	
int	Error code.

User's Notes

Because Zoltan places no restrictions on the content or length of GIDs, hashing does not guarantee a balanced distribution of objects in the distributed directory. Note also, the worst case behavior of a hash table lookup is very bad (essentially becoming a linear search). Fortunately, the average behavior is very good! The user may specify their own hash function via Zoltan_DD_Set_Hash_Fn to improve performance.

This software module is built on top of the Zoltan Communications functions for efficiency. Improvements to the communications library will automatically benefit the distributed directory.

FUTURE:

> The C99 capability for variable length arrays would significantly simplify many of these following routines. (It eliminates the malloc/free calls for temporary storage. This helps prevent memory leaks.) Other C99 features may also improve code readability. The "inline" capability can potentially improve performance.
>
> The distributed directory should be implemented via threads. However, MPI is not fully thread aware, yet.

[Table of Contents | Next: Examples of Zoltan Usage | Previous: Unstructured Communication Utilities]

Examples of Zoltan Usage

Examples for each part of the Zoltan library are provided:

- General use of Zoltan
- Load-balancing calling sequence
- Data migration calling sequences
- Query functions for a simple application

General Usage Example

An example of general Zoltan usage is included below. This is a C language example. Similar C++ examples may be found in the *examples* directory.

In this example, **Zoltan_Initialize** is called using the *argc* and *argv* arguments to the main program. Then a pointer to a Zoltan structure is returned by the call to **Zoltan_Create**. In this example, all processors will be used by Zoltan, as **MPI_COMM_WORLD** is passed to **Zoltan_Create** as the communicator.

Several application query functions are then registered with Zoltan through calls to **Zoltan_Set_Fn**. Parameters are set through calls to **Zoltan_Set_Param**. The application then performs in computations, including making calls to Zoltan functions and utilities.

Before its execution ends, the application frees memory used by Zoltan by calling **Zoltan_Destroy**.

```
/* Initialize the Zoltan library */
struct Zoltan_Struct *zz;
float version;
...
Zoltan_Initialize(argc, argv, &version);
zz = Zoltan_Create(MPI_COMM_WORLD);

/* Register query functions. */
Zoltan_Set_Fn(zz, ZOLTAN_NUM_GEOM_FN_TYPE,
          (void (*)()) user_return_dimension, NULL);
Zoltan_Set_Fn(zz, ZOLTAN_GEOM_FN_TYPE,
          (void (*)()) user_return_coords, NULL);
Zoltan_Set_Fn(zz, ZOLTAN_NUM_OBJ_FN_TYPE,
          (void (*)()) user_return_num_node, NULL);
Zoltan_Set_Fn(zz, ZOLTAN_OBJ_LIST_FN_TYPE,
          (void (*)()) user_return_owned_nodes, NULL);

/* Set some Zoltan parameters. */
Zoltan_Set_Param(zz, "debug_level", "4");

/* Perform application computations, call Zoltan, etc. */
...

/* Free Zoltan data structure before ending application. */
Zoltan_Destroy (&zz);
```

Typical calling sequence for general usage of the Zoltan library.

```
! Initialize the Zoltan library
type(Zoltan_Struct), pointer :: zz
real(Zoltan_FLOAT) version
integer(Zoltan_INT) ierr
...
ierr = Zoltan_Initialize(version) ! without argc and argv
zz => Zoltan_Create(MPI_COMM_WORLD)

! Register load-balancing query functions.
! omit data = C NULL
ierr = Zoltan_Set_Fn(zz, ZOLTAN_NUM_GEOM_FN_TYPE,
user_return_dimension)
ierr = Zoltan_Set_Fn(zz, ZOLTAN_GEOM_FN_TYPE,
user_return_coords)
ierr = Zoltan_Set_Fn(zz, ZOLTAN_NUM_OBJ_FN_TYPE,
user_return_num_node)
ierr = Zoltan_Set_Fn(zz, ZOLTAN_OBJ_LIST_FN_TYPE,
user_return_owned_nodes)

! Set some Zoltan parameters.
ierr = Zoltan_Set_Param(zz, "debug_level", "4")

! Perform application computations, call Zoltan, etc.
...

! Free Zoltan data structure before ending application.
call Zoltan_Destroy(zz)
```

Fortran version of general usage example.

[Table of Contents | Next: Load-Balancing Example | Previous: Examples of Library Usage]

Load-Balancing Example

An example of the typical calling sequence for load balancing using Zoltan in a finite element application is shown in the figure below. An application first selects a load-balancing algorithm by setting the *LB_METHOD* parameter with **Zoltan_Set_Param**. Next, other parameter values are set by calls to **Zoltan_Set_Param**. After some computation, load balancing is invoked by calling **Zoltan_LB_Partition**. The results of the load balancing include the number of nodes to be imported and exported to the processor, lists of global and local IDs of the imported and exported nodes, and source and destination processors of the imported and exported nodes. A returned argument of **Zoltan_LB_Partition** is tested to see whether the new decomposition differs from the old one. If the decompositions differ, some sort of data migration is needed to establish the new decomposition; the details of migration are not shown in this figure but will be addressed in the migration examples. After the data migration is completed, the arrays of information about imported and exported nodes returned by **Zoltan_LB_Partition** are freed by a call to **Zoltan_LB_Free_Part**.

```
char *lb_method;
int new, num_imp, num_exp, *imp_procs, *exp_procs;
int *imp_to_part, *exp_to_part;
int num_gid_entries, num_lid_entries;
ZOLTAN_ID_PTR imp_global_ids, exp_global_ids;
ZOLTAN_ID_PTR imp_local_ids, exp_local_ids;

/* Set load-balancing method. */
read_load_balancing_info_from_input_file(&lb_method);
Zoltan_Set_Param(zz, "LB_METHOD", lb_method);

/* Reset some load-balancing parameters. */
Zoltan_Set_Param(zz, "RCB_Reuse", "TRUE");

/* Perform computations */
...
/* Perform load balancing */
Zoltan_LB_Partition(zz,&new,&num_gid_entries,&num_lid_entries,

&num_imp,&imp_global_ids,&imp_local_ids,&imp_procs,&imp_to_part,

&num_exp,&exp_global_ids,&exp_local_ids,&exp_procs,&exp_to_part);
if (new)
  perform_data_migration(...);

/* Free memory allocated for load-balancing results by Zoltan
```

```
library */
Zoltan_LB_Free_Part(&imp_global_ids, &imp_local_ids, &imp_procs,
&imp_to_part);
Zoltan_LB_Free_Part(&exp_global_ids, &exp_local_ids, &exp_procs,
&exp_to_part);
...
```

Typical calling sequence for performing load balancing with the Zoltan library.

```
character(len=3) lb_method
logical new
integer(Zoltan_INT) num_imp, num_exp
integer(Zoltan_INT) num_gid_entries, num_lid_entries
integer(Zoltan_INT), pointer :: imp_procs(:), exp_procs(:)
integer(Zoltan_INT), pointer :: imp_global_ids(:),
exp_global_ids(:) ! global IDs
integer(Zoltan_INT), pointer :: imp_local_ids(:),
exp_local_ids(:) ! local IDs
integer(Zoltan_INT) ierr

! Set load-balancing method.
lb_method = "RCB"
ierr = Zoltan_Set_Param(zz, "LB_METHOD", lb_method)

! Reset some load-balancing parameters
ierr = Zoltan_Set_Param(zz, "RCB_Reuse", "TRUE")

! Perform computations
...
! Perform load balancing
ierr =
Zoltan_LB_Partition(zz,new,num_gid_entries,num_lid_entries, &
       num_imp,imp_global_ids,imp_local_ids, &
       imp_procs,imp_to_part, &
       num_exp,exp_global_ids,exp_local_ids, &
       exp_procs,exp_to_part)
if (new) then
  perform_data_migration(...)
endif

! Free memory allocated for load-balancing results by Zoltan
library
ierr = Zoltan_LB_Free_Part(imp_global_ids, imp_local_ids,
imp_procs, imp_to_part);
ierr = Zoltan_LB_Free_Part(exp_global_ids, exp_local_ids,
```

```
exp_procs, exp_to_part);
...
```

Fortran version of the load-balancing example.

[Table of Contents | Next: Migration Examples | Previous: General Usage Example]

Migration Examples

Data migration using Zoltan's migration tools can be accomplished in two different ways:

auto-migration, or
user-guided migration.

The choice of migration method depends upon the complexity of the application's data. For some applications, only the objects used in balancing must be migrated; no auxiliary data structures must be moved. Particle simulations are examples of such applications; load balancing is based on the number of particles per processor, and only the particles and their data must be moved to establish the new decomposition. For such applications, Zoltan's auto-migration tools can be used. Other applications, such as finite element methods, perform load balancing on, say, the nodes of the finite element mesh, but nodes that are moved to new processors also need to have their connected elements moved to the new processors, and migrated elements may also need "ghost" nodes (i.e., copies of nodes assigned to other processors) to satisfy their connectivity requirements on the new processor. This complex data migration requires a more user-controlled approach to data migration than the auto-migration capabilities Zoltan can provide.

Auto-Migration Example

In the figure below, an example of the load-balancing calling sequence for a particle simulation using Zoltan's auto-migration tools is shown. The application requests auto-migration by turning on the **AUTO_MIGRATE** option through a call to **Zoltan_Set_Param** and registers functions to pack and unpack a particle's data. During the call to **Zoltan_LB_Partition**, Zoltan computes the new decomposition and, using calls to the packing and unpacking query functions, automatically migrates particles to their new processors. The application then frees the arrays returned by **Zoltan_LB_Partition** and can continue computation without having to perform any additional operations for data migration.

```
/* Tell Zoltan to automatically migrate data for the application.
*/
Zoltan_Set_Param(zz, "AUTO_MIGRATE", "TRUE");

/* Register additional functions for packing and unpacking data
*/
/* by migration tools. */
Zoltan_Set_Fn(zz, ZOLTAN_OBJ_SIZE_FN_TYPE,
          (void (*)()) user_return_particle_data_size, NULL);
Zoltan_Set_Fn(zz, ZOLTAN_PACK_OBJ_FN_TYPE,
          (void (*)()) user_pack_particle_data, NULL);
Zoltan_Set_Fn(zz, ZOLTAN_UNPACK_OBJ_FN_TYPE,
          (void (*)()) user_unpack_particle_data, NULL);
...
/* Perform computations */
...
/* Perform load balancing AND automatic data migration! */
Zoltan_LB_Partition(zz,&new,&num_gid_entries,&num_lid_entries,

&num_imp,&imp_global_ids,&imp_local_ids,&imp_procs,&imp_to_part,
```

```
&num_exp,&exp_global_ids,&exp_local_ids,&exp_procs,&exp_to_part);
/* Free memory allocated for load-balancing results by Zoltan */
Zoltan_LB_Free_Part(&imp_global_ids, &imp_local_ids, &imp_procs,
&imp_to_part);
Zoltan_LB_Free_Part(&exp_global_ids, &exp_local_ids, &exp_procs,
&exp_to_part);
...
```

Typical calling sequence for using the migration tools' auto-migration capability with the dynamic load-balancing tools.

User-Guided Migration Example

In the following figure, an example of user-guided migration using Zoltan's migration tools for a finite element application is shown. Several migration steps are needed to completely rebuild the application's data structures for the new decomposition. On each processor, newly imported nodes need copies of elements containing those nodes. Newly imported elements, then, need copies of "ghost" nodes, nodes that are in the element but are assigned to other processors. Each of these entities (nodes, elements, and ghost nodes) can be migrated in separate migration steps using the functions provided in the migration tools. First, the assignment of nodes to processors returned by **Zoltan_LB_Partition** is established. Query functions that pack and unpack nodes are registered and **Zoltan_Migrate** is called using the nodal decomposition returned from **Zoltan_LB_Partition**. **Zoltan_Migrate** packs the nodes to be exported, sends them to other processors, and unpacks nodes received by a processor. The packing routine *migrate_node_pack* includes with each node a list of the element IDs for elements containing that node. The unpacking routine *migrate_node_unpack* examines the list of element IDs and builds a list of requests for elements the processor needs but does not already store. At the end of the nodal migration, each processor has a list of element IDs for elements that it needs to support imported nodes but does not already store. Through a call to **Zoltan_Invert_Lists**, each processor computes the list of elements it has to send to other processors to satisfy their element requests. Packing and unpacking routines for elements are registered, and **Zoltan_Migrate** is again used to move element data to new processors. Requests for ghost nodes can be built within the element packing and unpacking routines, and calls to **Zoltan_Invert_Lists** and **Zoltan_Migrate**, with node packing and unpacking, satisfy requests for ghost nodes. In all three phases of migration, the migration tools handle communication; the application is responsible only for packing and unpacking data and for building the appropriate request lists.

```
/* Assume Zoltan returns a decomposition of the */
/* nodes of a finite element mesh. */
Zoltan_LB_Partition(zz,&new,&num_gid_entries,&num_lid_entries,
    &num_imp,&imp_global_ids,&imp_local_ids,&imp_procs,&imp_to_part,
    &num_exp,&exp_global_ids,&exp_local_ids,&exp_procs,&exp_to_part);

/* Migrate the nodes as directed by the results of Zoltan_LB_Partition. */
/* While unpacking nodes, build list of requests for elements needed */
/* to support the imported nodes.*/
Zoltan_Set_Fn(zz, ZOLTAN_OBJ_SIZE_FN_TYPE,
           (void (*)()) migrate_node_size, NULL);
Zoltan_Set_Fn(zz, ZOLTAN_PACK_OBJ_FN_TYPE,
           (void (*)()) migrate_pack_node, NULL);
Zoltan_Set_Fn(zz, ZOLTAN_UNPACK_OBJ_FN_TYPE,
           (void (*)()) migrate_unpack_node, NULL);
```

```
Zoltan_Migrate(zz,num_import,imp_global_ids,imp_local_ids,imp_procs,imp_to_part,
        num_export,exp_global_ids,exp_local_ids,exp_procs,exp_to_part);

/* Prepare for migration of requested elements. */
Zoltan_Set_Fn(zz, ZOLTAN_PACK_OBJ_FN_TYPE,
           (void (*)()) migrate_pack_element, NULL);
Zoltan_Set_Fn(zz, ZOLTAN_UNPACK_OBJ_FN_TYPE,
           (void (*)()) migrate_unpack_element, NULL);
Zoltan_Set_Fn(zz, ZOLTAN_OBJ_SIZE_FN_TYPE,
           (void (*)()) migrate_element_size, NULL);

/* From the request lists, a processor knows which elements it needs */
/* to support the imported nodes; it must compute which elements to */
/* send to other processors. */
Zoltan_Invert_Lists(zz, Num_Elt_Requests, Elt_Requests_Global_IDs,
           Elt_Requests_Local_IDs, Elt_Requests_Procs, Elt_Requests_to_Part,
           &num_tmp_exp, &tmp_exp_global_ids,
           &tmp_exp_local_ids, &tmp_exp_procs, &tmp_exp_to_part);

/* Processor now knows which elements to send to other processors. */
/* Send the requested elements. While unpacking elements, build */
/* request lists for "ghost" nodes needed by the imported elements. */
Zoltan_Migrate(zz, Num_Elt_Requests, Elt_Requests_Global_IDs,
       Elt_Requests_Local_IDs, Elt_Requests_Procs, Elt_Request_to_Part,
       num_tmp_exp_objs, tmp_exp_global_ids,
       tmp_exp_local_ids, tmp_exp_procs, tmp_exp_to_part);

/* Repeat process for "ghost" nodes. */
...
```

Typical calling sequence for user-guided use of the migration tools in Zoltan.

[Table of Contents | Next: Query-Function Examples | Previous: Load-Balancing Example]

Query-Function Examples

Examples of query functions provided by a simple application are included below. The general-interface examples include a simple implementation of **ZOLTAN_GEOM_FN** and **ZOLTAN_OBJ_LIST_FN** query functions and variants of the simple implementation that exploit local identifiers and data pointers. Migration examples for packing and unpacking objects are also included. Robust error checking is not included in the routines; application developers should include more explicit error checking in their query functions.

All the examples use a mesh data structure consisting of nodes in the mesh. these nodes are the objects passed to Zoltan. A node is described by its 3D coordinates and a global ID number that is unique across all processors. The type definitions for the mesh and node data structures used in the examples are included below.

```
/* Node data structure. */
/* A node consists of its 3D coordinates and */
/* an ID number that is unique across all processors. */
struct Node_Type {
  double Coordinates[3];
  int Global_ID_Num;
};

/* Mesh data structure. */
/* Mesh consists of an array of nodes and */
/* the number of nodes owned by the processor. */
struct Mesh_Type {
  struct Node_Type Nodes[MAX_NODES];
  int Number_Owned;
};
```

Data types for the query-function examples.

```
! Node data structure.
! A node consists of its 3D coordinates and
! an ID number that is unique across all processors.
type Node_Type
  real(Zoltan_DOUBLE) :: Coordinates(3)
  integer(Zoltan_INT) :: Global_ID_Num
end type Node_Type


! Mesh data structure.
! Mesh consists of an array of nodes and
! the number of nodes owned by the processor.
type Mesh_Type
  type(Node_Type) :: Nodes(MAX_NODES)
  integer(Zoltan_INT) :: Number_Owned
end type Mesh_Type
```

Data types for the Fortran query-function examples.

General Interface Query Function Examples

In the following examples, **ZOLTAN_OBJ_LIST_FN** and **ZOLTAN_GEOM_FN** query functions are implemented for an application using the mesh and node data structures described above. The nodes are the objects passed to Zoltan.

Through a call to **Zoltan_Set_Fn**, the function *user_return_owned_nodes* is registered as the **ZOLTAN_OBJ_LIST_FN** query function. It returns global and local identifiers for each node owned by a processor.

The function *user_return_coords* is registered as a **ZOLTAN_GEOM_FN** query function. Given the global and local identifiers for a node, this function returns the node's coordinates. All the examples exploit the local identifier to quickly locate nodal data. If such an identifier is not available in an application, a search using the global identifier can be performed.

The Basic Example includes the simplest implementation of the query routines. In the query routines, it uses global application data structures and a local numbering scheme for the local identifiers. The User-Defined Data Pointer Example uses only local application data structures; this model is useful if the application does not have global data structures or if objects from more than one data structure are to be passed to Zoltan. Differences between the latter example and the Basic Example are shown in red.

Basic Example

In the simplest example, the query functions access the application data through a global data structure (*Mesh*) representing the mesh. In the calls to **Zoltan_Set_Fn**, no pointers to application data are registered with the query function (i.e., the *data* pointer is not used). A node's local identifier is an integer representing the index in the *Mesh.Nodes* array of the node. The local identifier is set to the index's value in *user_return_owned_nodes*. It is used to access the global *Mesh.Nodes* array in *user_return_coords*.

```
/* in application's program file */
#include "zoltan.h"

/* Declare a global Mesh data structure. */
struct Mesh_Type Mesh;

main()
{
...
    /* Indicate that local and global IDs are one integer
each. */
    Zoltan_Set_Param(zz, "NUM_GID_ENTRIES", "1");
    Zoltan_Set_Param(zz, "NUM_LID_ENTRIES", "1");

    /* Register query functions. */
    /* Do not register a data pointer with the functions; */
    /* the global Mesh data structure will be used. */
    Zoltan_Set_Fn(zz, ZOLTAN_GEOM_FN_TYPE,
              (void (*)()) user_return_coords, NULL);
    Zoltan_Set_Fn(zz, ZOLTAN_OBJ_LIST_FN_TYPE,
              (void (*)()) user_return_owned_nodes, NULL);
...
}

void user_return_owned_nodes(void *data,
     int num_gid_entries, int num_lid_entries,
     ZOLTAN_ID_PTR global_ids, ZOLTAN_ID_PTR local_ids,
     int wgt_dim, float *obj_wgts,
     int *ierr)
{
int i;
    /* return global node numbers as global_ids. */
    /* return index into Nodes array for local_ids. */
    for (i = 0; i < Mesh.Number_Owned; i++){
        global_ids[i*num_gid_entries] =
Mesh.Nodes[i].Global_ID_Num;
        local_ids[i*num_lid_entries] = i;
    }
    *ierr = ZOLTAN_OK;
}

void user_return_coords(void *data,
```

```
     int num_gid_entries, int num_lid_entries,
     ZOLTAN_ID_PTR global_id, ZOLTAN_ID_PTR local_id,
     double *geom_vec, int *ierr)
{
    /* use local_id to index into the Nodes array. */
    geom_vec[0] = Mesh.Nodes[local_id[0]].Coordinates[0];
    geom_vec[1] = Mesh.Nodes[local_id[0]].Coordinates[1];
    geom_vec[2] = Mesh.Nodes[local_id[0]].Coordinates[2];
    *ierr = ZOLTAN_OK;
}
```

Example of general interface query functions (simplest implementation).

```
! in application's program file

module Global_Mesh_Data
! Declare a global Mesh data structure.
   type(Mesh_Type) :: Mesh
end module

program query_example_1
use zoltan
...
    ! Indicate that local and global IDs are one integer
each.
    ierr = Zoltan_Set_Param(zz, "NUM_GID_ENTRIES", "1");
    ierr = Zoltan_Set_Param(zz, "NUM_LID_ENTRIES", "1");

    ! Register query functions.
    ! Do not register a data pointer with the functions;
    ! the global Mesh data structure will be used.
    ierr = Zoltan_Set_Fn(zz, ZOLTAN_GEOM_FN_TYPE,
user_return_coords)
    ierr = Zoltan_Set_Fn(zz, ZOLTAN_OBJ_LIST_FN_TYPE,
user_return_owned_nodes)
...
end program

subroutine user_return_owned_nodes(data, &
    num_gid_entries, num_lid_entries, &
    global_ids, local_ids, wgt_dim, obj_wgts, ierr)
use zoltan
use Global_Mesh_Data
integer(Zoltan_INT) :: data(1) ! dummy declaration, do not use
integer(Zoltan_INT), intent(in) :: num_gid_entries,
```

```
num_lid_entries
integer(Zoltan_INT), intent(out) :: global_ids(*),
local_ids(*)
integer(Zoltan_INT), intent(in) :: wgt_dim
real(Zoltan_FLOAT), intent(out) :: obj_wgts(*)
integer(Zoltan_INT), intent(out) :: ierr
integer i
    ! return global node numbers as global_ids.
    ! return index into Nodes array for local_ids.
    do i = 1, Mesh%Number_Owned
        global_ids(1+(i-1)*num_gid_entries) = &
            Mesh%Nodes(i)%Global_ID_Num
        local_ids(1+(i-1)*num_lid_entries) = i
    end do
    ierr = ZOLTAN_OK
end subroutine

subroutine user_return_coords(data, num_gid_entries,
num_lid_entries, &
    global_id, local_id, geom_vec, ierr)
use zoltan
use Global_Mesh_Data
integer(Zoltan_INT) :: data(1) ! dummy declaration, do not use
integer(Zoltan_INT), intent(in) :: num_gid_entries,
num_lid_entries
integer(Zoltan_INT), intent(in) :: global_id(*), local_id(*)
real(Zoltan_DOUBLE), intent(out) :: geom_vec(*)
integer(Zoltan_INT), intent(out) :: ierr
    ! use local_id to index into the Nodes array.
    geom_vec(1:3) = Mesh%Nodes(local_id(1))%Coordinates
    ierr = ZOLTAN_OK
end subroutine
```

Fortran example of general interface query functions (simplest implementation).

User-Defined Data Pointer Example

In this example, the address of a local mesh data structure is registered with the query functions for use by those functions. This change eliminates the need for a global mesh data structure in the application. The address of the local data structure is included as an argument in calls to **Zoltan_Set_Fn**. This address is then used in *user_return_owned_nodes* and *user_return_coords* to provide data for these routines. It is cast to the *Mesh_Type* data type and accessed with local identifiers as in the Basic Example. Differences between this example and the Basic Example are shown in red.

This model is useful when the application does not have a global data structure that can be accessed by the query functions. It can also be used for operations on different data structures. For example, if an application had more than one mesh, load balancing could be performed separately on each mesh without having different query routines for each mesh. Calls to **Zoltan_Set_Fn** would define which mesh should be balanced, and the query routines would access the mesh currently designated by the **Zoltan_Set_Fn** calls.

```
/* in application's program file */
#include "zoltan.h"

main()
{
/* declare a local mesh data structure. */
struct Mesh_Type mesh;
...
    /* Indicate that local and global IDs are one integer
each. */
    Zoltan_Set_Param(zz, "NUM_GID_ENTRIES", "1");
    Zoltan_Set_Param(zz, "NUM_LID_ENTRIES", "1");

    /* Register query functions. */
    /* Register the address of mesh as the data pointer. */
    Zoltan_Set_Fn(zz, ZOLTAN_GEOM_FN_TYPE,
              (void (*)()) user_return_coords, &mesh);
    Zoltan_Set_Fn(zz, ZOLTAN_OBJ_LIST_FN_TYPE,
              (void (*)()) user_return_owned_nodes, &mesh);
...
}

void user_return_owned_nodes(void *data,
     int num_gid_entries, int num_lid_entries,
     ZOLTAN_ID_PTR global_ids, ZOLTAN_ID_PTR local_ids,
     int wgt_dim, float *obj_wgts,
     int *ierr)
{
int i;
/* cast data pointer to type Mesh_Type. */
struct Mesh_Type *ptr = (struct Mesh_Type *) data;

    /* return global node numbers as global_ids. */
    /* return index into Nodes array for local_ids. */
    for (i = 0; i < ptr->Number_Owned; i++) {
        global_ids[i*num_gid_entries] =
ptr->Nodes[i].Global_ID_Num;
        local_ids[i*num_lid_entries] = i;
    }
    *ierr = ZOLTAN_OK;
}
```

```
void user_return_coords(void *data,
     int num_gid_entries, int num_lid_entries,
     ZOLTAN_ID_PTR global_id, ZOLTAN_ID_PTR local_id,
     double *geom_vec, int *ierr)
{
/* cast data pointer to type Mesh_Type. */
struct Mesh_Type *ptr = (struct Mesh_Type *) data;

    /* use local_id to address the requested node. */
    geom_vec[0] = ptr->Nodes[local_id[0]].Coordinates[0];
    geom_vec[1] = ptr->Nodes[local_id[0]].Coordinates[1];
    geom_vec[2] = ptr->Nodes[local_id[0]].Coordinates[2];
    *ierr = ZOLTAN_OK;
}
```

Example of general interface query functions using the application-defined data pointer.

```
/* included in file zoltan_user_data.f90 */
! User defined data type as wrapper for Mesh
type Zoltan_User_Data_1
   type(Mesh_type), pointer :: ptr
end type Zoltan_User_Data_1
```

```
! in application's program file

program query_example_3
use zoltan
! declare a local mesh data structure and a User_Data to point
to it.
type(Mesh_Type), target :: mesh
type(Zoltan_User_Data_1) data
...
    ! Indicate that local and global IDs are one integer
each.
    ierr = Zoltan_Set_Param(zz, "NUM_GID_ENTRIES", "1");
    ierr = Zoltan_Set_Param(zz, "NUM_LID_ENTRIES", "1");

    ! Register query functions.
    ! Use the User_Data variable to pass the mesh data
    data%ptr => mesh
    ierr = Zoltan_Set_Fn(zz, ZOLTAN_GEOM_FN_TYPE,
user_return_coords, data)
    ierr = Zoltan_Set_Fn(zz, ZOLTAN_OBJ_LIST_FN_TYPE,
                     user_return_owned_nodes, data)
```

```
...
end program

subroutine user_return_owned_nodes(data, &
    num_gid_entries, num_lid_entries, &
    global_ids, local_ids, wgt_dim, obj_wgts, ierr)
use zoltan
type(Zoltan_User_Data_1) :: data
integer(Zoltan_INT), intent(in) :: num_gid_entries,
num_lid_entries
integer(Zoltan_INT), intent(out) :: global_ids(*),
local_ids(*)
integer(Zoltan_INT), intent(in) :: wgt_dim
real(Zoltan_FLOAT), intent(out) :: obj_wgts(*)
integer(Zoltan_INT), intent(out) :: ierr
integer i
type(Mesh_Type), pointer :: Mesh

   ! extract the mesh from the User_Data argument
    Mesh => data%ptr

    ! return global node numbers as global_ids.
    ! return index into Nodes array for local_ids.
    do i = 1, Mesh%Number_Owned
        global_ids(1+(i-1)*num_gid_entries) = &
            Mesh%Nodes(i)%Global_ID_Num
        local_ids(1+(i-1)*num_lid_entries) = i
    end do
    ierr = ZOLTAN_OK
end subroutine

subroutine user_return_coords(data, global_id, local_id, &
     geom_vec, ierr)
use zoltan
type(Zoltan_User_Data_1) :: data
integer(Zoltan_INT), intent(in) :: num_gid_entries,
num_lid_entries
integer(Zoltan_INT), intent(in) :: global_id(*), local_id(*)
real(Zoltan_DOUBLE), intent(out) :: geom_vec(*)
integer(Zoltan_INT), intent(out) :: ierr
type(Mesh_Type), pointer :: Mesh

   ! extract the mesh from the User_Data argument
    Mesh => data%ptr

    ! use local_id to index into the Nodes array.
    geom_vec(1:3) = Mesh%Nodes(local_id(1))%Coordinates
    ierr = ZOLTAN_OK
```

```
end subroutine
```

Fortran example of general interface query functions using the application-defined data pointer.

Migration Examples

Packing and Unpacking Data

Simple migration query functions for the Basic Example are included below. These functions are used by the migration tools to move nodes among the processors. The functions *user_size_node*, *user_pack_node*, and *user_unpack_node* are registered through calls to **Zoltan_Set_Fn**. Query function *user_size_node* returns the size (in bytes) of data representing a single node. Query function *user_pack_node* copies a given node's data into the communication buffer *buf*. Query function *user_unpack_node* copies a data for one node from the communication buffer *buf* into the *Mesh.Nodes* array on its new processor.

These query routines are simple because the application does not dynamically allocate memory for each node. Such dynamic allocation would have to be accounted for in the ZOLTAN_OBJ_SIZE_FN, ZOLTAN_PACK_OBJ_FN, and ZOLTAN_UNPACK_OBJ_FN routines.

```
main()
{
...
    /* Register migration query functions. */
    /* Do not register a data pointer with the functions; */
    /* the global Mesh data structure will be used. */
    Zoltan_Set_Fn(zz, ZOLTAN_OBJ_SIZE_FN_TYPE,
              (void (*)()) user_size_node, NULL);
    Zoltan_Set_Fn(zz, ZOLTAN_PACK_OBJ_FN_TYPE,
              (void (*)()) user_pack_node, NULL);
    Zoltan_Set_Fn(zz, ZOLTAN_UNPACK_OBJ_FN_TYPE,
              (void (*)()) user_unpack_node, NULL);
...
}

int user_size_node(void *data,
    int num_gid_entries, int num_lid_entries,
    ZOLTAN_ID_PTR global_id, ZOLTAN_ID_PTR local_id, int
*ierr)
{
/* Return the size of data associated with one node. */
/* This case is simple because all nodes have the same size.
*/
    *ierr = ZOLTAN_OK;
```

```
    return(sizeof(struct Node_Type));
}

void user_pack_node(void *data,
     int num_gid_entries, int num_lid_entries,
     ZOLTAN_ID_PTR global_id, ZOLTAN_ID_PTR local_id,
     int dest_proc, int size, char *buf, int *ierr)
{
/* Copy the specified node's data into buffer buf. */
struct Node_Type *node_buf = (struct Node_Type *) buf;

    *ierr = ZOLTAN_OK;
    node_buf->Coordinates[0] =
Mesh.Nodes[local_id[0]].Coordinates[0];
    node_buf->Coordinates[1] =
Mesh.Nodes[local_id[0]].Coordinates[1];
    node_buf->Coordinates[2] =
Mesh.Nodes[local_id[0]].Coordinates[2];
    node_buf->Global_ID_Num =
Mesh.Nodes[local_id[0]].Global_ID_Num;
}

void user_unpack_node(void *data, int num_gid_entries,
     ZOLTAN_ID_PTR global_id, int size,
     char *buf, int *ierr)
{
/* Copy the node data in buf into the Mesh data structure. */
int i;
struct Node_Type *node_buf = (struct Node_Type *) buf;

    *ierr = ZOLTAN_OK;
    i = Mesh.Number_Owned;
    Mesh.Number_Owned = Mesh.Number_Owned + 1;
    Mesh.Nodes[i].Coordinates[0] = node_buf->Coordinates[0];
    Mesh.Nodes[i].Coordinates[1] = node_buf->Coordinates[1];
    Mesh.Nodes[i].Coordinates[2] = node_buf->Coordinates[2];
    Mesh.Nodes[i].Global_ID_Num = node_buf->Global_ID_Num;
}
```

Example of migration query functions for the Basic Example.

[Table of Contents | Next: FORTRAN Interface | Previous: Migration Examples]

FORTRAN Interface

The Fortran interface for Zoltan is a Fortran 90 interface designed similar to the Fortran 90 Bindings for OpenGL [Mitchell]. There is no FORTRAN 77 interface; however, FORTRAN 77 applications can use Zoltan by adding only a few Fortran 90 statements, which are fully explained in the section on FORTRAN 77, provided that vendor-specific extensions are not heavily used in the application. This section describes how to build the Fortran interface into the Zoltan library, how to call Zoltan from Fortran applications, and how to compile Fortran applications that use Zoltan. Note that the capitalization used in this section is for clarity and need not be adhered to in the application code, since Fortran is case insensitive.

FORTRAN: Compiling Zoltan

To include the Fortran interface in the Zoltan library, use the **YES_FORTRAN** parameter in the make statement; for example

gmake YES_FORTRAN=1 ZOLTAN_ARCH=<*platform*> zoltan

Before compiling the library, make sure that the application's *zoltan_user_data.f90* has been placed in the *Zoltan/fort/* directory.

FORTRAN: Compiling Applications

To compile a Fortran application using the Zoltan library, the module information files must be made available to most compilers during the compilation phase. Module information files are files generated by the compiler to provide module information to program units that **USE** the module. They usually have suffixes like *.mod* or *.M*. The module information files for the modules in the Zoltan library are located in the *Obj_<platform>* subdirectory. Most Fortran 90 compilers have a compile line flag to specify directories to be searched for module information files, typically "-I"; check the documentation for your compiler. If your compiler does not have such a flag, you will have to copy the module information files to the directory of the application (or use symbolic links).

The Fortran interface is built into the same library file as the rest of Zoltan, which is found during the compiler link phase with *-lzoltan*. Thus an example compilation line would be

f90 -I<path to Zoltan>/*Obj_<platform>* application.f90 -lzoltan

---

FORTRAN API

The Fortran interface for each Zoltan Interface Function and Application-Registered Query Function is given along with the C interface. This section contains some general information about the design and use of the Fortran interface.

Names

All procedure, variable, defined constant and structure names are identical to those in the C interface, except that in Fortran they are case insensitive (either upper or lower case letters can be used).

Zoltan module

MODULE *zoltan* provides access to all entities in Zoltan that are of use to the application, including kind type parameters, named constants, procedures, and derived types. Any program unit (e.g., main program, module, external subroutine) that needs access to an entity from Zoltan must contain the statement

USE zoltan

near the beginning.

Numeric types

The correspondence between Fortran and C numeric types is achieved through the use of kind type parameters. In most cases, the default kind for a Fortran type will match the corresponding C type, but this is not guaranteed. To insure portability of the application code, it is highly recommended that the following kind type parameters be used in the declaration of all variables and constants that will be passed to a Zoltan procedure:

C	**Fortran**
int	INTEGER(KIND=Zoltan_INT)
float	REAL(KIND=Zoltan_FLOAT)
double	REAL(KIND=Zoltan_DOUBLE)

Note that "KIND=" is optional in declaration statements. The kind number for constants can be attached to the constant, e.g., 1.0_Zoltan_DOUBLE.

Structures

For any struct in the C interface to Zoltan, e.g. **Zoltan_Struct**, there is a corresponding derived type in the Fortran interface. Variables of this type are declared as demonstrated below:

```
TYPE(Zoltan_Struct) :: zz
```

In the Fortran interface, the internal components of the derived type are PRIVATE and not accessible to the application. However, the application simply passes these variables around, and never needs to access the internal components.

Global and local IDs

While the C implementation uses arrays of unsigned integers to represent global and local IDs, the Fortran interface uses arrays of integers, as unsigned integers are not available in Fortran. Thus, each ID is represented as an array (possibly of size 1) of integers. Applications that use other data types for their IDs can convert between their data types and Zoltan's in the application-registered query functions.

Query function data

Zoltan_Set_Fn allows the application to pass a pointer to data that will subsequently be passed to the query function being registered. From Fortran this is an optional argument, or can be one of several types. In the simplest cases, an intrinsic array containing the data will be sufficient. For these cases, data can be an assumed size array of type INTEGER(Zoltan_INT), REAL(Zoltan_FLOAT) or REAL(Zoltan_DOUBLE). When the argument is omitted in the call to the registration function, a data argument will still be passed to the query function. This should be declared as an assumed size array of type INTEGER(Zoltan_INT) and never used.

For more complicated situations, the application may need to pass data in a user-defined type. The strong type checking of Fortran does not allow passing an arbitrary type without modifying the Fortran interface for each desired type. So the Fortran interface provides a type to be used for this purpose, **Zoltan_User_Data_1**. Since different types of data may need to be passed to different query functions, four such types are provided, using the numerals 1, 2, 3 and 4 as the last character in the name of the type. These types are defined by the application in *zoltan_user_data.f90*. If not needed, they must be defined, but can be almost empty as in *fort/zoltan_user_data.f90*.

The application may use these types in any appropriate way. If desired, it can define these types to contain the application's data and use the type throughout the application. But it is anticipated that in most cases, the desired type already exists in the application, and the **Zoltan_User_Data_x** types will be used as "wrapper types," containing one or more pointers to the existing types. For example,

```
TYPE mesh
    ! an existing data type with whatever defines a mesh
END TYPE mesh
```

```
TYPE Zoltan_User_Data_2
    TYPE(mesh), POINTER :: ptr
END TYPE Zoltan_User_Data_2
```

The application would then set the pointer to the data before calling Zoltan_Set_Fn:

```
TYPE(mesh) :: meshdata
TYPE(Zoltan_User_Data_2) :: query_data
TYPE(Zoltan_Struct) :: zz
INTEGER(Zoltan_INT), EXTERNAL :: num_obj_func ! not required for module procedures

query_data%ptr => meshdata
ierr = Zoltan_Set_Fn(zz,ZOLTAN_NUM_OBJ_FN_TYPE,num_obj_func,query_data)
```

Note that the existing data type must be available when **Zoltan_User_Data_x** is defined. Therefore it must be defined either in *zoltan_user_data.f90* or in a module that is compiled before *zoltan_user_data.f90* and **USE**d by MODULE *zoltan_user_data*. For an example that uses a wrapper type, see *fdriver/zoltan_user_data.f90*.

[Table of Contents | Next: FORTRAN 77 | Previous: FORTRAN--Compiling Applications

FORTRAN 77

There is no FORTRAN 77 interface for Zoltan; however, an existing FORTRAN 77 application can be compiled by a Fortran 90 compiler provided it does not use vendor specific extensions (unless the same extensions are supported by the Fortran 90 compiler), and the application can use Zoltan's Fortran 90 interface with a minimal amount of Fortran 90 additions. This section provides details of the Fortran 90 code that must be added.

When building the Zoltan library, use the file *fort/zoltan_user_data.f90* for *zoltan_user_data.f90*. This assumes that DATA in a call to **ZOLTAN_SET_FN** is either omitted (you can omit arguments that are labeled OPTIONAL in the Fortran API) or an array of type INTEGER, REAL or DOUBLE PRECISION (REAL*4 and REAL*8 might be acceptable). If a more complicated set of data is required (for example, two arrays), then it should be made available to the query functions through COMMON blocks.

To get access to the interface, each program unit (main program, subroutine or function) that calls a Zoltan routine must begin with the statement

    USE ZOLTAN

and this should be the first statement after the program, subroutine or function statement (before the declarations).

The pointer to the Zoltan structure returned by **ZOLTAN_CREATE** should be declared as

    TYPE(ZOLTAN_STRUCT), POINTER :: ZZ

(you can use a name other than ZZ if you wish).

To create the structure, use a pointer assignment statement with the call to **ZOLTAN_CREATE**:

    ZZ => ZOLTAN_CREATE(COMMUNICATOR)

Note that the assignment operator is "=>". If ZZ is used in more than one procedure, then put it in a COMMON block. It cannot be passed as an argument unless the procedure interfaces are made "explicit." (Let's not go there.)

The eight import and export arrays passed to **ZOLTAN_LB_PARTITION** (and other procedures) must be pointers. They should be declared as, for example,

    INTEGER, POINTER :: IMPORT_GLOBAL_IDS(:)

Note that the double colon after POINTER is required, and the dimension must be declared as "(:)" with a colon. Like ZZ, if they are used in more than one procedure, pass them through a COMMON block, not as an argument.

Except in the unlikely event that the default kinds of intrinsic types do not match the C intrinsic types, you do not have to use the kind type parameters **Zoltan_INT**, etc. It is also not necessary to include the INTENT attribute in the declarations of the query functions, so they can be simplified to, for example,

    SUBROUTINE GET_OBJ_LIST(DATA, GLOBAL_IDS, LOCAL_IDS, WGT_DIM,
    OBJ_WGTS, IERR)

```
INTEGER DATA(*),GLOBAL_IDS(*),LOCAL_IDS(*),WGT_DIM,IERR
REAL OBJ_WGTS(*)
```

to be more consistent with a FORTRAN 77 style.

[Table of Contents | Next: FORTRAN--System-Specific Remarks | Previous: FORTRAN API

FORTRAN: System-Specific Remarks

System-specific details of the FORTRAN interface are included below.

The mention of specific products, trademarks, or brand names is for purposes of identification only. Such mention is not to be interpreted in any way as an endoresement or certification of such products or brands by the National Institute of Standards and Technology or Sandia National Laboratories. All trademarks mentioned herein belong to their respective owners.

MPICH

As of version 1.1.2, the MPICH implementation of MPI is not completely "Fortran 90 friendly." Only one problem was encountered during our tests: the reliance on command line arguments. MPICH uses command line arguments during the start-up process, even if the application does not. Command line arguments are not standard in Fortran, so although most compilers offer it as an extension, each compiler has its own method of handling them. The problem arises when one Fortran compiler is specified during the build of MPICH and another Fortran compiler is used for the application. This should not be a problem on systems where there is only one Fortran compiler, or where multiple Fortran compilers are compatible (for example, FORTRAN 77 and Fortran 90 compilers from the same vendor). If your program can get past the call to MPI_Init, then you do not have this problem.

To solve this problem, build MPICH in such a way that it does not include the routines for *iargc* and *getarg* (I have been able to do this by using the -f95nag flag when configuring MPICH), and then provide your own versions of them when you link the application. Some versions of these routines are provided in *fdriver/farg_**.

Pacific Sierra

Pacific Sierra Research (PSR) Vastf90 is not currently supported due to bugs in the compiler with no known workarounds. It is not known when or if this compiler will be supported.

NASoftware

N.A.Software FortranPlus is not currently supported due to problems with the query functions. We anticipate that this problem can be overcome, and support will be added soon.

C++ Interface

The C++ interface to the Zoltan library is contained in the header files listed below. Each header file defines one class. Each class represents a Zoltan data structure and the functions that operate on that data structure. The class methods in the header files call functions in the Zoltan C library. So to use the C++ interface from your application, include the appropriate header file and link with the Zoltan C library.

header file	class
include/zoltan_cpp.h	**Zoltan**, representing a load balancing instance
Utilities/Communication/zoltan_comm_cpp.h	**Zoltan_Comm**, representing an unstructured communication instance
Utilities/DDirectory/zoltan_dd_cpp.h	**Zoltan_DD**, representing a distributed directory instance
Utilities/Timer/zoltan_timer_cpp.h	**Zoltan_Timer**, representing a timer instance

More detailed information about the interface may be found in the Zoltan Developer's Guide.

Simple examples of the use of the interface may be found in the *examples/CPP* directory. A more complete example is the test driver zCPPdrive. The source code for this test driver is in the *driver* directory.

A note on declaring application registered query functions from a C++ application may be found in the section titled Application-Registered Query Functions.

Two peculiarities of the wrapping of Zoltan with C++ classes are mentioned here:

1. You must call the C language function *Zoltan_Initialize* before using the C++ interface to the Zoltan library. This function should only be called once. Due to design choices, the C++ interface maintains no global state that is independent of any instantiated objects, so it does not know if the function has been called or not. Therefore, the C++ wrappers do not call *Zoltan_Initialize* for you.
2. It is preferable to allocate **Zoltan** objects dynamically so you can explicitly delete them before your application exits. (**Zoltan** objects allocated instead on the stack will be deleted automatically at the completion of the scope in which they were created.) The reason is that the **Zoltan** destructor calls Zoltan_Destroy(), which makes an MPI call to free the communicator in use by the **Zoltan** object. However the MPI destructor may have been called before the **Zoltan** destructor. In this case you would receive an error while your application is exiting.

This second point is illustrated in the good and bad example below.

```
int main(int argc, char *argv[])
{
 MPI::Init(argc, argv);
 int rank = MPI::COMM_WORLD.Get_rank();
 int size = MPI::COMM_WORLD.Get_size();

 //Initialize the Zoltan library with a C language call
 float version;
 Zoltan_Initialize(argc, argv, &version);

 //Dynamically create Zoltan object.
 Zoltan *zz = new Zoltan(MPI::COMM_WORLD);
 zz->Set_Param("LB_METHOD", "RCB");

 //Several lines of code would follow, working with zz

 //Explicitly delete the Zoltan object
 delete zz;
 MPI::Finalize();
 }
```

Good example, Zoltan object is dynamically allocated and explicity deleted before exit.

```
int main(int argc, char *argv[])
{
Zoltan zz;

 MPI::Init(argc, argv);
 int rank = MPI::COMM_WORLD.Get_rank();
 int size = MPI::COMM_WORLD.Get_size();

 //Initialize the Zoltan library with a C language call
 float version;
 Zoltan_Initialize(argc, argv, &version);

 zz.Set_Param("LB_METHOD", "RCB");

 //Several lines of code would follow, working with zz

 MPI::Finalize();
 }
```

Bad example, the MPI destructor may execute before the Zoltan destructor at process exit.

[Table of Contents | Next: Backward Compatibility | Previous: FORTRAN: System-Specific Remarks Examples

Backward Compatibility with Previous Versions of Zoltan

As new features have been added to Zoltan, backward compatibility with previous versions of Zoltan has been maintained. Thus, users of previous versions of Zoltan can upgrade to a new version **without changing their application source code**. Modifications to application source code are needed **only** if the applications use new Zoltan functionality.

Enhancements to the Zoltan interface are described below.

Backward Compatibility: Versions 1.5 and higher

The ability to generate more partitions than processors was added to Zoltan in version 1.5. Thus, Zoltan's partitioning and migration routines were enhanced to return and use both partition assignments and processor assignments. New interface and query functions were added to support this additional information. All former Zoltan parameters apply to the new functions as they did to the old; new parameters *NUM_GLOBAL_PARTITIONS* and *NUM_LOCAL_PARTITIONS* apply only to the new functions.

The table below lists the Zoltan function that uses both partition and processor information, along with the analogous function that returns only processor information. Applications requiring only one partition per processor can use either version of the functions.

Function with Partition and Processor info (v1.5 and higher)	Function with only Processor info (v1.3 and higher)
Zoltan_LB_Partition	Zoltan_LB_Balance
Zoltan_LB_Point_PP_Assign	Zoltan_LB_Point_Assign
Zoltan_LB_Box_PP_Assign	Zoltan_LB_Box_Assign
Zoltan_Invert_Lists	Zoltan_Compute_Destinations
Zoltan_Migrate	Zoltan_Help_Migrate
ZOLTAN_PRE_MIGRATE_PP_FN	ZOLTAN_PRE_MIGRATE_FN
ZOLTAN_MID_MIGRATE_PP_FN	ZOLTAN_MID_MIGRATE_FN
ZOLTAN_POST_MIGRATE_PP_FN	ZOLTAN_POST_MIGRATE_FN

To continue using the v1.3 partition functions, no changes to C or Fortran90 applications are needed. Zoltan interfaces from versions earlier than 1.3 are also still supported (see below), requiring no changes to application programs.

To use the new v1.5 partitioning functions:

- C users must include file *zoltan.h* in their applications and edit their applications to use the appropriate new functions.
- Fortran90 users must put user-defined data types in *zoltan_user_data.f90* and edit their applications to use the appropriate new functions. The new partitioning functions do not work with user-defined data types in *lb_user_const.f90*.

Backward Compatibility: Versions 1.3 and higher

Versions of Zoltan before version 1.3 used a different naming convention for the Zoltan interface and query functions. All functions in Zoltan v.1.3 and above are prefixed with **Zoltan_**; earlier versions were prefixed with **LB_**.

Zoltan versions 1.3 and above maintain backward compatibility with the earlier Zoltan interface. Thus, applications that used earlier versions of Zoltan can continue using Zoltan **without changing their source code**.

Only two changes are needed to build the application with Zoltan v.1.3 and higher:

- All Zoltan include files are now in directory *Zoltan/include*. Thus, application include paths must point to this directory.
  (Previously, include files were in *Zoltan/lb*.)
- Applications link with Zoltan now by specifying only *-lzoltan*.
  (Previously, applications had to link with *-lzoltan -lzoltan_comm -lzoltan_mem*.)

While it is not necessary for application developers to modify their source code to use Zoltan v.1.3 and above, those who want to update their source code should do the following in their application source files:

- Replace Zoltan calls and constants (**LB_***) with new names. The new names can be found through the index below.
- C programs: Include file *zoltan.h* instead of *lbi_const.h*.
- F90 programs: Put user-defined data types in file *zoltan_user_data.f90* instead of *lb_user_const.f90*.

Backward Compatilibity Index for Interface and Query Functions

Name in Earlier Zoltan Versions	Name in Zoltan Version 1.3 and higher
LB_BORDER_OBJ_LIST_FN	ZOLTAN_BORDER_OBJ_LIST_FN

LB_Balance	Zoltan_LB_Balance
LB_Box_Assign	Zoltan_LB_Box_Assign
LB_CHILD_LIST_FN	ZOLTAN_CHILD_LIST_FN
LB_CHILD_WEIGHT_FN	ZOLTAN_CHILD_WEIGHT_FN
LB_COARSE_OBJ_LIST_FN	ZOLTAN_COARSE_OBJ_LIST_FN
LB_Compute_Destinations	Zoltan_Compute_Destinations
LB_Create	Zoltan_Create
LB_Destroy	Zoltan_Destroy
LB_EDGE_LIST_FN	ZOLTAN_EDGE_LIST_FN
LB_Eval	Zoltan_LB_Eval
LB_FIRST_BORDER_OBJ_FN	ZOLTAN_FIRST_BORDER_OBJ_FN
LB_FIRST_COARSE_OBJ_FN	ZOLTAN_FIRST_COARSE_OBJ_FN
LB_FIRST_OBJ_FN	ZOLTAN_FIRST_OBJ_FN
LB_Free_Data	Zoltan_LB_Free_Data
LB_GEOM_FN	ZOLTAN_GEOM_FN
LB_Help_Migrate	Zoltan_Help_Migrate
LB_Initialize	Zoltan_Initialize
LB_MID_MIGRATE_FN	ZOLTAN_MID_MIGRATE_FN
LB_NEXT_BORDER_OBJ_FN	ZOLTAN_NEXT_BORDER_OBJ_FN
LB_NEXT_COARSE_OBJ_FN	ZOLTAN_NEXT_COARSE_OBJ_FN
LB_NEXT_OBJ_FN	ZOLTAN_NEXT_OBJ_FN
LB_NUM_BORDER_OBJ_FN	ZOLTAN_NUM_BORDER_OBJ_FN
LB_NUM_CHILD_FN	ZOLTAN_NUM_CHILD_FN
LB_NUM_COARSE_OBJ_FN	ZOLTAN_NUM_COARSE_OBJ_FN
LB_NUM_EDGES_FN	ZOLTAN_NUM_EDGES_FN

LB_NUM_GEOM_FN	ZOLTAN_NUM_GEOM_FN
LB_NUM_OBJ_FN	ZOLTAN_NUM_OBJ_FN
LB_OBJ_LIST_FN	ZOLTAN_OBJ_LIST_FN
LB_OBJ_SIZE_FN	ZOLTAN_OBJ_SIZE_FN
LB_PACK_OBJ_FN	ZOLTAN_PACK_OBJ_FN
LB_POST_MIGRATE_FN	ZOLTAN_POST_MIGRATE_FN
LB_PRE_MIGRATE_FN	ZOLTAN_PRE_MIGRATE_FN
LB_Point_Assign	Zoltan_LB_Point_Assign
LB_Set_Fn	Zoltan_Set_Fn
LB_Set_<*lb_fn_type*>_Fn	Zoltan_Set_<*zoltan_fn_type*>_Fn
LB_Set_Method	Zoltan_Set_Param with parameter *LB_METHOD*
LB_Set_Param	Zoltan_Set_Param
LB_UNPACK_OBJ_FN	ZOLTAN_UNPACK_OBJ_FN

[Table of Contents | Next: References | Previous: C++ Interface]

References

1. "ALEGRA -- A Framework for Large Strain Rate Physics." http://sherpa.sandia.gov/9231home/alegra/alegra-frame.html
2. S. Attaway, T. Barragy, K. Brown, D. Gardner, B. Hendrickson, S. Plimpton and C. Vaughan. "Transient Solid Dynamics Simulations on the Sandia/Intel Teraflop Computer." *Proceedings of SC'97*, San Jose, CA, November, 1997. (Finalist for the Gordon Bell Prize.) U. Catalyurek and C. Aykanat, "Hypergraph-partitioning-based decomposition for parallel sparse matrix vector multiplication", *IEEE Trans. Parallel Dist. Systems*, v. 10, no. 7, (1999) pp. 673--693.
3. P. Baehmann, S. Wittchen, M. Shephard, K. Grice, and M. Yerry. "Robust geometrically based automatic two-dimensional mesh generation." *Intl. J. Numer. Meths. Engrg.*, *24* (1987) 1043-1078.
4. E.G. Boman, D. Bozdag, U. Catalyurek, A.H. Gebremedhin and F. Manne. "A Scalable Parallel Graph Coloring Algorithm for Distributed Memory Computers". *Proceedings of Euro-Par'05*, Lisbon, Portugal, August, 2005.
5. D. Bozdag, U. Catalyurek, A.H. Gebremedhin, F. Manne, E.G. Boman and F. Ozguner. "A Parallel Distance-2 Graph Coloring Algorithm for Distributed Memory Computers". *Proceedings of HPCC'05*, Sorrento, Italy, September, 2005.
6. M. Berger and S. Bokhari. "A partitioning strategy for nonuniform problems on multiprocessors." *IEEE Trans. Computers*, C-36 (1987) 570-580.
7. K.D. Devine, E.G. Boman, R. Heaphy, R.H. Bisseling, U.V. Catalyurek. "Parallel Hypergraph Partitioning for Scientific Computing", Proc. of IPDPS'06, Rhodos, Greece, April 2006.
8. K. Devine, G. Hennigan, S. Hutchinson, A. Salinger, J. Shadid, and R. Tuminaro. "High Performance MP Unstructured Finite Element Simulation of Chemically Reacting Flows." *Proceedings of SC'97*, San Jose, CA, November, 1997. (Finalist for the Gordon Bell Prize.)
9. H.C. Edwards. *A parallel infrastructure for scalable adaptive finite element methods and its application to least squares C^(inf)* collocation. Ph.D. Dissertation, Univ. of Texas at Austin, May, 1997.
10. J. Flaherty, R. Loy, M. Shephard, B. Szymanski, J. Teresco and L. Ziantz. "Adaptive local refinement with octree load-balancing for the parallel solution of three-dimensional conservation laws." *J. Parallel Distrib. Comput.*, *47* (1998) 139-152.
11. L. Gervasio. "Final Report." Summer project report, Internal Memo, Department 9103, Sandia National Laboratories, August, 1998.
12. B. Hendrickson and K. Devine. "Dynamic load balancing in computational mechanics." *Comp. Meth. Appl. Mech. Engrg.*, v. 184 (#2-4), p. 485-500, 2000.
13. B. Hendrickson and T.G. Kolda. "Partitioning rectangular and structurally nonsymmetric sparse matrices for parallel computation", *SIAM J. on Sci. Comp.*, v. 21, no. 6, 2001, pp. 2048-2072.
14. B. Hendrickson and R. Leland. "The Chaco user's guide, version 2.0." Tech. Rep. SAND 94-2692, Sandia National Laboratories, Albuquerque, NM, October, 1994. http://www.cs.sandia.gov/CRF/chac.html

15. G. Karypis and V. Kumar. "ParMETIS: Parallel graph partitioning and sparse matrix ordering library." Tech. Rep. 97-060, Department of Computer Science, Univ. of Minnesota, 1997. http://www-users.cs.umn.edu/~karypis/metis/parmetis/
16. R. Loy. *Adaptive local refinement with octree load-balancing for the parallel solution of three-dimensional conservation laws*. Ph. D. Dissertation, Dept. of Computer Science, Rensselaer Polytechnic Institute, May 1998.
17. S. Mitchell and S. Vavasis. "Quality mesh generation in three dimensions." *Proc. 8th ACM Symposium on Computational Geometry*, ACM (1992) 212-221.
18. W. F. Mitchell. "A Fortran 90 Interface for OpenGL: Revised January 1998" NISTIR 6134 (1998). http://math.nist.gov/~mitchell/papers/nistir6134.ps.gz
19. W. F. Mitchell. "The K-way Refinement Tree Partitioning Method for Adaptive Grids." http://math.nist.gov/~mitchell/papers/parcomp.ps.gz
20. "MPSalsa: Massively Parallel Numerical Methods for Advanced Simulation of Chemically Reacting Flows." http://www.cs.sandia.gov/CRF/MPSalsa/
21. A. Patra and J. T. Oden. "Problem decomposition for adaptive hp-finite element methods." *J. Computing Systems in Engrg.*, 6 (1995).
22. J. Pilkington and S. Baden. "Partitioning with space-filling curves." Tech. Rep. CS94-349, Dept. of Computer Science and Engineering, Univ. of California, San Diego, CA, 1994.
23. M. Shephard and M. Georges. "Automatic three-dimensional mesh generation by the finite octree technique." *Intl. J. Numer. Meths. Engrg.*, 32 (1991) 709-749.
24. V. E. Taylor and B. Nour-Omid. "A Study of the Factorization Fill-in for a Parallel Implementation of the Finite Element Method." *Intl. J. Numer. Meths. Engrg.*, 37 (1994) 3809-3823.
25. C. Walshaw. "JOSTLE mesh partitioning software", http://www.gre.ac.uk/jostle/
26. C. Walshaw, M. Cross, and M. Everett. "Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes", *J. Par. Dist. Comp.*, 47(2) 102-108, 1997.
27. M. Warren and J. Salmon. "A parallel hashed octree n-body algorithm." *Proc. Supercomputing `93*, Portland, OR, November 1993.
28. R. D. Williams. "Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency, Practice, and Experience*, 3(5), 457-481, 1991.

[Table of Contents | Next: Index of Interface and Query Functions | Previous: Backward Compatibility]

Index of Interface and Query Functions

[Table of Contents | Previous: References | Zoltan Home Page]